

C#

编程语言详解

C# Programming Language

[美] Anders Hejlsberg Scott Wiltamuth Peter Golde 著
张晓坤 谭立平 车树良 译

Microsoft
.net
Development
Series

.NET 技术大系

C#编程语言详解

The C# Programming Language

[美] Anders Hejlsberg Scott Wiltamuth Peter Golde 著

张晓坤 谭立平 车树良 译

电子工业出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

C#是一门简单、现代、优雅、面向对象、类型安全、平台独立的组件编程语言，是.NET 的关键性语言，也是整个.NET 平台的基础，它使程序员能快速地为新一代 Microsoft .NET 平台开发出应用程序。全书以通俗易懂的语言，精辟丰富的实例，从对 C#的简介开始，全面讲解了 C#编程语言规范以及各个层面的特性，内容包括 C#的词法结构、类型、变量、表达式、类、结构、不安全代码、泛型，等等。

本书内容翔实，实用性强，适合作为高等院校学生学习编程语言的教材，也是希望深入探索 C#编程知识的广大程序开发人员绝佳的技术参考书。

Simplified Chinese edition copyright © 2004 by PEARSON EDUCATION ASIA LIMITED and Publishing House of Electronics Industry.

The C# Programming Language, First Edition, ISBN: 0-321-15491-6 by Anders Hejlsberg, Copyright © 2004 by Microsoft Corporation.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Sun Microsystems, Inc..

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书中文简体字翻译版由电子工业出版社和 Pearson Education 培生教育出版亚洲有限公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2003-8568

图书在版编目 (CIP) 数据

C#编程语言详解 / (美) 海吉斯博格 (Hejlsberg, A.) 等著; 张晓坤, 谭立平, 车树良译.

北京: 电子工业出版社, 2004.9

(.NET 技术大系)

书名原文: The C# Programming Language

ISBN 7-121-00228-0

I. C... II. ①海...②张...③谭...④车... III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2004) 第 080838 号

责任编辑: 朱沐红 高洪霞

印 刷: 北京民族印刷厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 31.5 字数: 733 千字

印 次: 2004 年 9 月第 1 次印刷

印 数: 5 000 册 定价: 55.00 元

凡购买电子工业出版社的图书, 如有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系。联系电话: (010) 68279077。质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

译者序

.NET 正式推出（2002 年 2 月 13 日 Microsoft 正式发布 .NET）也不过两年多的时间，而市面上关于 C# 的各类参考书可谓琳琅满目，C# 程序员阵营也日益庞大。是什么原因使 C# 脱颖而出，成为编程语言的新贵？

我们追溯到 1995 年的春天，James Gosling 等人公开发布了一种名叫 Java 的编程语言。Java 在 Internet 的推动下，短短几年就受到开发人员的挚爱。网络、分布式使可移植性、安全等因素显得尤为重要。传统的开发模型和运行环境，在面对 Internet 时显得束手无策。Java 语法虽然酷似 C++，但本质与 C++ 相差甚远，它能够真正实现跨平台和可移植性。于是，Sun, IBM, Oracle, BEA, Borland 等 IT 领域的巨无霸都纷纷推出各自的 Java 产品，甚至 Microsoft 也有自己的 Java 产品。一时间，Java 成了众星捧月的香饽饽。而 Sun 与 Microsoft 也因为 Java 陷入了旷日持久的官司中。

我曾经设想，如果 Java 不属于 Sun 公司，或者 Sun 公司很慷慨，那么可能就不会有现在的 C#。不过，自由竞争是值得鼓励和尊敬的。在 Java 的赞扬声中也夹杂着不少反对意见，例如，Java 的 I/O 非常复杂，Swing 与 AWT 更是引来许多抨击，性能方面也是可圈可点的，等等。

面对 Java、Linux 等的竞争与冲击，Microsoft 酝酿着一个巨大的革命性的计划，这就是 .NET。Microsoft 将 .NET 视为数字化未来的一个远景和平台，正如其平台战略副总裁 Sanjay Parthasarathy 先生所说的那样^{译注}：

“ .NET 要做什么？它要加速向分布式计算的转移。

“ .NET 要做什么？它要拉动三个杠杆，分别是

- ◆ 一切都要为 Web 服务；
- ◆ 聚合与集成 Web 服务；
- ◆ 提供简单而令人神往的用户体验。”

C# 是完全基于 CLR 的编程语言。相信大家对其主创人员 Anders Hejlsberg 并不陌生。

我想大多数的程序员应该是通过 Delphi 知道了 Anders Hejlsberg 这个名字的。Hejlsberg 第一次在软件舞台亮相是在 20 世纪 80 年代早期，他为 MS-DOS 和 CP/M 设计了 Pascal 编译器。当时，Borland 还只是一家小公司，但它独具慧眼，很快雇用了他，并买下了他的编译器，改名为 Turbo Pascal。加盟 Borland 之后，Hejlsberg 继续开发 Turbo Pascal，并最终带领他的小组设计了 Turbo Pascal 的替代品：Delphi。1996 年，在为 Borland 工作了

^{译注} 摘自 “The Simplest Way to Define .NET（定义 .NET 的最简单方法）”，详情请访问 http://www.microsoft.com/china/net/define_net.asp

13 年之后, Hejlsberg 加入了 Microsoft。从最初的 Visual J++ 和 Windows Foundation Classes (WFC) 的架构师, 到 C# 的首席设计师和 .NET Framework 的关键参与者, 他算得上是一个传奇的人物。而本书的作者之一就是这位蜚声软件界的奇才——Anders Hejlsberg。

这本书是迄今为止市面上关于 C# 的语法、语义和设计最完整和精确的一本书。由于本书主要阐述的是 C# 编程语言规范, 因此, 全书的风格比较严谨和客观, 没有美国人惯有的那种调侃与幽默。全书共分三大部分, 第一部分关于 C# 语言的语法和基本特征, 面面俱到, 细致入微; 第二部分介绍 C# 2.0 引入的新特征, 包括泛型、匿名方法、迭代器和不完整类型; 第三部分“附录”介绍文档注释、文法等。

本书第 1~9 章介绍了 C# 的基本特征, 包括词法、类型、变量、转换、表达式、语句和命名空间等基本元素。

第 10~17 章介绍 C# 的面向对象特性, 包括类 (System.Object 是一切类的根)、结构、数组 (所有的数组隐式派生于 System.Array 类)、枚举 (所有枚举隐式派生于 System.Enum)、委托 (面向对象的指针 [即类型安全])、异常 (一种结构化的、统一的和类型安全的处理机制) 和特性 (使程序员能够为程序中定义的各种实体附加一些说明性信息, 可以在运行时读取)。

第 18 章介绍不安全代码, unsafe 模式下的不安全代码, 应该是 C# 的一个特色。

第 19 章介绍文档注释, 一种可以使用 XML 文本的特殊标注注释的语法, 借文档生成器生成文档文件的机制。

第 20~23 章介绍 C# 2.0 提供的编译时构件, 包括泛型、匿名方法、迭代器和不完整类型 (请于 Microsoft 的官方网站下载最新的 .NET Framework 1.2 和 .NET sdk 1.2)。

感谢 Anders Hejlsberg 等为我们带来一次神奇的体验!

由于水平有限, 错误之处在所难免, 恳请读者批评指正。

译者
2004 年 6 月

序 言

C#项目的启动差不多是在5年前——1998年12月，其目标在于创建一个简单、现代、面向对象及类型安全的编程语言，用在全新的并被命名为.NET的平台上。从那时候起，C#已经走过了漫长的道路。到目前为止，成千上万的程序员在使用C#语言，ECMA^{译注1}和ISO/IEC^{译注2}已经分别对它进行了标准化。此外，C#第二个版本的开发也接近尾声，它具有几个显著的新特点。

本书是C#编程语言完整的技术规范。它总共分为三个部分。第一部分“C# 1.0”包括第1~18章，描述了C# 1.0语言，它与Visual Studio .NET2002和2003一起发布。第二部分“C# 2.0”包括第19~23章，描述了C# 2.0的4个显著特点：泛型、匿名方法、迭代器和不完整类型。第三部分“附录”描述了文档注释，并且总结了第一部分中词法和句法的语法。截止到本书写作时，C# 2.0已经接近beta测试了。由于C# 2.0仍然在开发过程中，因此，第二部分所描述的新特点可能与最终版有所不同。不过，我们期望其中的变动尽量很小。

有许多人参与到C#语言的创建工作中。C# 1.0的语言设计小组包括Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Peter Sollich和Eric Gunnerson。对于C# 2.0，语言设计小组包括Anders Hejlsberg, Peter Golde, Peter Hallam, Shon Katzenberger, Todd Proebsting和Anson Horton。此外，C#和.NET公共语言运行库中泛型的设计和实现是基于“Gyro”原型，它是由Microsoft Research的Don Syme和Andrew Kennedy构建的。

我们不可能向所有影响C#设计的人员一一致谢，但还是要表示感激之情。在“真空”中是不能设计出好的语言的：我们拥有庞大而热情的用户基础，他们的反馈是我们的无价之宝。

C#已经并继续成为我们最具有挑战性和激动人心的项目。我们希望它能够喜欢使用C#，就如同我们热衷于创建它一样。

Anders Hejlsberg
Scott Wiltamuth
Peter Golde

2003年8月，西雅图

^{译注1} ECMA：欧洲计算机生产商协会[European Computer Manufacturers Association (ECMA)]，始建于1961年，致力于制订国际信息和通讯标准以及消费类电子标准。

^{译注2} ISO/IEC：国际标准化组织(International Organization for Standardization, ISO)是世界上最主要的非政府间国际标准化机构，正式成立于1947年2月23日。它的前身是国家标准化协会国际联合会(ISA)和联合国际标准协调委员会(UNSCC)。ISO总部设在瑞士日内瓦，网址为<http://www.iso.ch>。国际电工委员会(International Electro technical Commission, IEC)的起源是1904年在美国圣路易召开的一次电气大会上通过的一项决议。根据这项决议，1906年成立了IEC，它是世界上成立最早的一个标准化国际机构，网址为<http://www.iec.ch/>。

目 录

第一篇 C#1.0	1
第 1 章 C#简介	3
1.1 Hello World	3
1.2 程序结构	4
1.3 类型和变量	6
1.4 表达式	8
1.5 语句	9
1.6 类和对象	12
1.6.1 成员	13
1.6.2 可访问性	14
1.6.3 基类	14
1.6.4 字段	15
1.6.5 方法	15
1.6.6 其他函数成员	21
1.7 结构	25
1.8 数组	27
1.9 接口	28
1.10 枚举	29
1.11 委托	31
1.12 特性	32
第 2 章 词法结构	34
2.1 程序	34
2.2 文法	34
2.2.1 文法表示法	34
2.2.2 词法文法	35
2.2.3 句法文法	35
2.3 词法分析	36
2.3.1 行结束符	36
2.3.2 空白	37
2.3.3 注释	37
2.4 标记	38
2.4.1 Unicode 字符转义序列	39
2.4.2 标识符	39
2.4.3 关键字	41

2.4.4	文本	42
2.4.5	运算符和标点	47
2.5	预处理指令	47
2.5.1	条件编译符号	48
2.5.2	预处理表达式	49
2.5.3	声明指令	50
2.5.4	条件编译指令	51
2.5.5	诊断指令	53
2.5.6	区域指令	54
2.5.7	行指令	55
第 3 章	基本概念	56
3.1	应用程序启动	56
3.2	应用程序终止	57
3.3	声明	57
3.4	成员	59
3.4.1	命名空间成员	59
3.4.2	结构成员	60
3.4.3	枚举成员	60
3.4.4	类成员	60
3.4.5	接口成员	60
3.4.6	数组成员	61
3.4.7	委托成员	61
3.5	成员访问	61
3.5.1	已声明可访问性	61
3.5.2	可访问域	62
3.5.3	实例成员的受保护访问	64
3.5.4	可访问性约束	65
3.6	签名和重载	66
3.7	范围	66
3.7.1	通过嵌套隐藏	69
3.7.2	通过继承隐藏	69
3.8	命名空间和类型名称	70
3.9	自动内存管理	72
3.10	执行顺序	75
第 4 章	类型	76
4.1	值类型	76
4.1.1	System.ValueType 类型	77
4.1.2	默认构造函数	77
4.1.3	结构类型	78

4.1.4	简单类型	78
4.1.5	整型	79
4.1.6	浮点型	80
4.1.7	decimal 类型	81
4.1.8	bool 类型	82
4.1.9	枚举类型	82
4.2	引用类型	82
4.2.1	类类型	83
4.2.2	对象类型	84
4.2.3	string 类型	84
4.2.4	接口类型	84
4.2.5	数组类型	84
4.2.6	委托类型	84
4.3	装箱和取消装箱	85
4.3.1	装箱转换	85
4.3.2	取消装箱转换	86
第 5 章	变量	87
5.1	变量类别	87
5.1.1	静态变量	87
5.1.2	实例变量	88
5.1.3	数组元素	88
5.1.4	值参数	88
5.1.5	引用参数	88
5.1.6	输出参数	89
5.1.7	局部变量	89
5.2	默认值	90
5.3	明确赋值	90
5.3.1	初始已赋值变量	91
5.3.2	初始未赋值变量	91
5.3.3	确定明确赋值的细则	91
5.4	变量引用	101
5.5	变量引用的原子性	101
第 6 章	转换	102
6.1	隐式转换	102
6.1.1	标识转换	102
6.1.2	隐式数值转换	102
6.1.3	隐式枚举转换	103
6.1.4	隐式引用转换	103
6.1.5	装箱转换	104

6.1.6	隐式常数表达式转换	104
6.1.7	用户定义的隐式转换	104
6.2	显式转换	104
6.2.1	显式数值转换	105
6.2.2	显式枚举转换	106
6.2.3	显式引用转换	106
6.2.4	取消装箱转换	107
6.2.5	用户定义的显式转换	107
6.3	标准转换	107
6.3.1	标准隐式转换	107
6.3.2	标准显式转换	108
6.4	用户定义的转换	108
6.4.1	允许的用户定义转换	108
6.4.2	用户定义的转换的计算	108
6.4.3	用户定义的隐式转换	109
6.4.4	用户定义的显式转换	110
第 7 章	表达式	112
7.1	表达式的分类	112
7.2	运算符	113
7.2.1	运算符的优先级和顺序关联性	113
7.2.2	运算符重载	114
7.2.3	一元运算符重载决策	115
7.2.4	二元运算符重载决策	116
7.2.5	候选用户定义运算符	116
7.2.6	数值提升	116
7.3	成员查找	118
7.4	函数成员	119
7.4.1	参数列表	120
7.4.2	重载决策	123
7.4.3	函数成员调用	125
7.5	基本表达式	126
7.5.1	文本	127
7.5.2	简单名称	127
7.5.3	带括号的表达式	128
7.5.4	成员访问	129
7.5.5	调用表达式	131
7.5.6	元素访问	132
7.5.7	this 访问	134
7.5.8	base 访问	134

7.5.9	后缀增量和后缀减量运算符	135
7.5.10	new 运算符	136
7.5.11	typeof 运算符	140
7.5.12	checked 和 unchecked 运算符	141
7.6	一元运算符	143
7.6.1	一元加运算符	143
7.6.2	一元减运算符	144
7.6.3	逻辑否定运算符	144
7.6.4	按位求补运算符	145
7.6.5	前缀增量和减量运算符	145
7.6.6	强制转换表达式	146
7.7	算术运算符	146
7.7.1	乘法运算符	147
7.7.2	除法运算符	148
7.7.3	余数运算符	149
7.7.4	加法运算符	150
7.7.5	减法运算符	152
7.8	移位运算符	153
7.9	关系和类型测试运算符	155
7.9.1	整数比较运算符	155
7.9.2	浮点比较运算符	156
7.9.3	小数比较运算符	157
7.9.4	布尔相等运算符	157
7.9.5	枚举比较运算符	157
7.9.6	引用类型相等运算符	158
7.9.7	字符串相等运算符	159
7.9.8	委托相等运算符	159
7.9.9	is 运算符	160
7.9.10	as 运算符	160
7.10	逻辑运算符	161
7.10.1	整数逻辑运算符	161
7.10.2	枚举逻辑运算符	162
7.10.3	布尔逻辑运算符	162
7.11	条件逻辑运算符	162
7.11.1	布尔条件逻辑运算符	163
7.11.2	用户定义的条件逻辑运算符	163
7.12	条件运算符	164
7.13	赋值运算符	165
7.13.1	简单赋值	165

7.13.2	复合赋值	167
7.13.3	事件赋值	168
7.14	表达式	168
7.15	常数表达式	168
7.16	布尔表达式	169
第 8 章	语句	170
8.1	结束点和可到达性	170
8.2	块	172
8.3	空语句	173
8.4	标记语句	173
8.5	声明语句	174
8.5.1	局部变量声明	174
8.5.2	局部常数声明	175
8.6	表达式语句	175
8.7	选择语句	176
8.7.1	if 语句	176
8.7.2	switch 语句	177
8.8	迭代语句	181
8.8.1	while 语句	181
8.8.2	do 语句	181
8.8.3	for 语句	182
8.8.4	foreach 语句	183
8.9	跳转语句	185
8.9.1	break 语句	186
8.9.2	continue 语句	187
8.9.3	goto 语句	187
8.9.4	return 语句	188
8.9.5	throw 语句	189
8.10	try 语句	190
8.11	checked 语句和 unchecked 语句	193
8.12	lock 语句	193
8.13	using 语句	194
第 9 章	命名空间	197
9.1	编译单元	197
9.2	命名空间声明	197
9.3	using 指令	199
9.3.1	using 别名指令	199
9.3.2	using 命名空间指令	201
9.4	命名空间成员	203

9.5 类型声明	204
第 10 章 类	205
10.1 类声明	205
10.1.1 类修饰符	205
10.1.2 类基规范	207
10.1.3 类体	208
10.2 类成员	208
10.2.1 继承	209
10.2.2 new 修饰符	210
10.2.3 访问修饰符	210
10.2.4 构成类型	210
10.2.5 静态成员和实例成员	211
10.2.6 嵌套类型	212
10.2.7 保留成员名称	215
10.3 常数	217
10.4 字段	218
10.4.1 静态字段和实例字段	220
10.4.2 只读字段	220
10.4.3 易失字段	221
10.4.4 字段初始化	223
10.4.5 变量初始值设定项	223
10.5 方法	226
10.5.1 方法参数	227
10.5.2 静态方法和实例方法	233
10.5.3 虚拟方法	233
10.5.4 重写方法	235
10.5.5 密封方法	237
10.5.6 抽象方法	237
10.5.7 外部方法	239
10.5.8 方法体	239
10.5.9 方法重载	240
10.6 属性	240
10.6.1 静态属性和实例属性	241
10.6.2 访问器	241
10.6.3 虚拟、密封、重写和抽象访问器	246
10.7 事件	247
10.7.1 类似字段的事件	249
10.7.2 事件访问器	250
10.7.3 静态事件和实例事件	252

10.7.4 虚拟、密封、重写和抽象访问器	252
10.8 索引器	252
10.9 运算符	256
10.9.1 一元运算符	257
10.9.2 二元运算符	258
10.9.3 转换运算符	259
10.10 实例构造函数	260
10.10.1 构造函数初始值设定项	261
10.10.2 实例变量初始值设定项	262
10.10.3 构造函数执行	262
10.10.4 默认构造函数	264
10.10.5 私有构造函数	264
10.10.6 可选的实例构造函数参数	265
10.11 静态构造函数	265
10.12 析构函数	267
第 11 章 结构	270
11.1 结构声明	270
11.1.1 结构修饰符	270
11.1.2 结构接口	271
11.1.3 结构体	271
11.2 结构成员	271
11.3 类和结构的区别	272
11.3.1 值语义	272
11.3.2 继承	273
11.3.3 赋值	273
11.3.4 默认值	273
11.3.5 装箱和取消装箱	274
11.3.6 this 的意义	274
11.3.7 字段初始值设定项	274
11.3.8 构造函数	275
11.3.9 析构函数	276
11.4 结构示例	276
11.4.1 数据库整数类型	276
11.4.2 数据库布尔类型	277
第 12 章 数组	280
12.1 数组类型	280
12.2 数组创建	281
12.3 数组元素访问	281
12.4 数组成员	281

12.5	数组协方差	281
12.6	数组初始值设定项	282
第 13 章	接口	284
13.1	接口声明	284
13.1.1	接口修饰符	284
13.1.2	基接口	285
13.1.3	接口体	285
13.2	接口成员	285
13.2.1	接口方法	286
13.2.2	接口属性	287
13.2.3	接口事件	287
13.2.4	接口索引器	287
13.2.5	接口成员访问	288
13.3	完全限定接口成员名	289
13.4	接口实现	290
13.4.1	显式接口成员实现	291
13.4.2	接口映射	293
13.4.3	接口实现继承	295
13.4.4	接口重新实现	297
13.4.5	抽象类和接口	298
第 14 章	枚举	299
14.1	枚举声明	299
14.2	枚举修饰符	300
14.3	枚举成员	300
14.4	System.Enum 类型	302
14.5	枚举值和运算	302
第 15 章	委托	303
15.1	委托声明	303
15.2	委托实例化	305
15.3	委托调用	306
第 16 章	异常	308
16.1	导致异常的原因	308
16.2	System.Exception 类	308
16.3	异常的处理方式	309
16.4	公共异常类	309
第 17 章	特性	311
17.1	特性类	311
17.1.1	特性用法	311
17.1.2	定位和命名参数	312

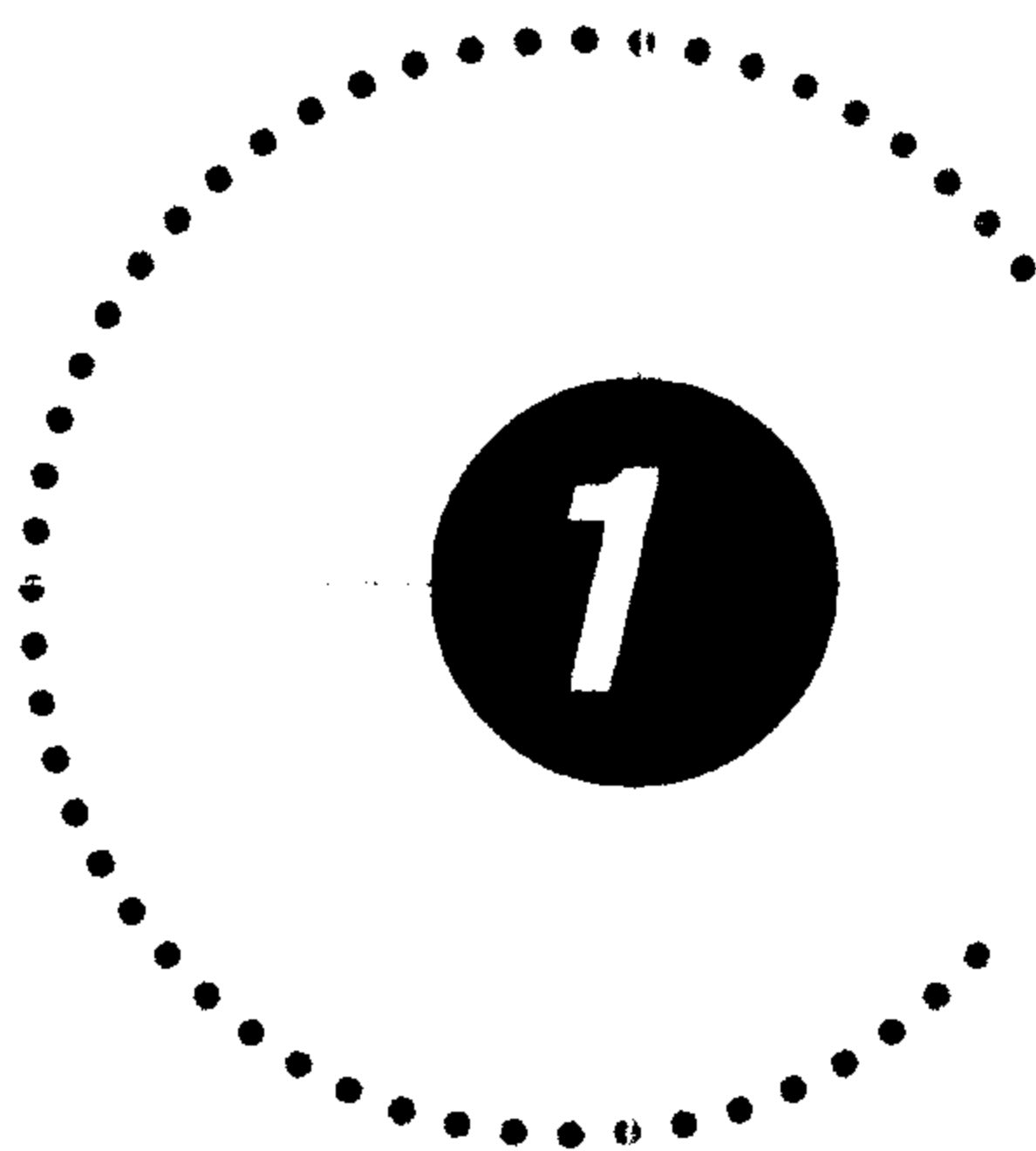
17.1.3 特性参数类型.....	313
17.2 特性专用化.....	314
17.3 特性实例.....	318
17.3.1 特性的编译.....	319
17.3.2 特性实例的运行时检索.....	319
17.4 保留特性.....	319
17.4.1 AttributeUsage 特性.....	320
17.4.2 Conditional 特性.....	320
17.4.3 Obsolete 特性.....	323
17.5 互操作的特性.....	324
17.5.1 与 COM 和 Win32 组件的互操作.....	324
17.5.2 与其他 .NET 语言的交互操作.....	324
第 18 章 不安全代码.....	325
18.1 不安全上下文.....	325
18.2 指针类型.....	328
18.3 固定变量和可移动变量.....	330
18.4 指针转换.....	331
18.5 表达式中的指针.....	332
18.5.1 指针间接寻址.....	332
18.5.2 指针成员访问.....	333
18.5.3 指针元素访问.....	334
18.5.4 address-of 运算符.....	335
18.5.5 指针增加和指针减小.....	335
18.5.6 指针算法.....	336
18.5.7 指针比较.....	337
18.5.8 sizeof 运算符.....	337
18.6 固定语句.....	338
18.7 堆栈分配.....	341
18.8 动态内存分配.....	342
第二篇 C# 2.0.....	345
第 19 章 C# 2.0 介绍.....	347
19.1 泛型.....	347
19.1.1 为什么使用泛型.....	347
19.1.2 创建和使用泛型.....	348
19.1.3 泛型类型实例化.....	349
19.1.4 约束.....	350
19.1.5 泛型方法.....	351
19.2 匿名方法.....	352

19.3	迭代器	354
19.4	不完整类型	358
第 20 章	泛型	360
20.1	泛型类声明	360
20.1.1	类型参数	360
20.1.2	实例类型	362
20.1.3	基类规范	362
20.1.4	泛型类的成员	363
20.1.5	泛型类中的静态字段	363
20.1.6	泛型类中的静态构造函数	364
20.1.7	访问受保护的成员	364
20.1.8	在泛型类中重载	365
20.1.9	参数数组方法和类型参数	366
20.1.10	重写和泛型类	366
20.1.11	泛型类中的运算符	367
20.1.12	泛型类中的嵌套类型	368
20.1.13	应用程序入口点	369
20.2	泛型结构声明	369
20.3	泛型接口声明	369
20.3.1	实现接口的惟一性	370
20.3.2	显式接口成员实现	370
20.4	泛型委托声明	371
20.5	构造类型	371
20.5.1	类型实参	373
20.5.2	开放类型和封闭类型	373
20.5.3	构造类型的基类和接口	373
20.5.4	构造类型的成员	374
20.5.5	构造类型的可访问性	375
20.5.6	转换	375
20.5.7	System.Nullable<T>类型	376
20.5.8	使用别名指令	376
20.5.9	特性	376
20.6	泛型方法	377
20.6.1	泛型方法签名	378
20.6.2	虚拟泛型方法	379
20.6.3	调用泛型方法	379
20.6.4	类型实参推断	379
20.6.5	语法歧义	381
20.6.6	对委托使用泛型方法	381

20.6.7 非泛型属性、事件、索引器或运算符	382
20.7 约束	382
20.7.1 满足约束	384
20.7.2 类型参数上的成员查找	385
20.7.3 类型参数和装箱	385
20.7.4 包含类型参数的转换	387
20.8 表达式和语句	388
20.8.1 默认值表达式	388
20.8.2 对象创建表达式	389
20.8.3 type of 运算符	389
20.8.4 引用相等运算符	390
20.8.5 is 运算符	390
20.8.6 as 运算符	390
20.8.7 异常语句	391
20.8.8 lock 语句	391
20.8.9 using 语句	391
20.8.10 foreach 语句	391
20.9 查找规则修订	392
20.9.1 命名空间和类型名称	392
20.9.2 成员查找	393
20.9.3 简单名称	395
20.9.4 成员访问	396
20.9.5 方法调用	397
20.9.6 委托创建表达式	399
20.10 右移语法改变	399
第 21 章 匿名方法	401
21.1 匿名方法表达式	401
21.2 匿名方法签名	401
21.3 匿名方法转换	402
21.4 匿名方法块	403
21.5 外部变量	404
21.5.1 捕获外部变量	404
21.5.2 局部变量实例化	405
21.6 匿名方法求值	407
21.7 委托实例相等性	407
21.8 明确赋值	408
21.9 方法组转换	408
21.10 实现示例	409

第 22 章 迭代器	412
22.1 迭代器块	412
22.1.1 枚举器接口	412
22.1.2 可枚举接口	412
22.1.3 yield 类型	412
22.1.4 this 访问	413
22.2 枚举对象	413
22.2.1 MoveNext 方法	414
22.2.2 Current 属性	415
22.2.3 Dispose 方法	415
22.3 可枚举对象	415
22.4 yield 语句	416
22.5 实现示例	418
第 23 章 不完整类型	422
23.1 不完整类型声明	422
23.1.1 特性	423
23.1.2 修饰符	423
23.1.3 类型参数和约束	423
23.1.4 基类	424
23.1.5 基接口	424
23.1.6 成员	425
23.2 名称绑定	425
第三篇 附录	427
附录 A 文档注释	429
附录 B 语法	447

第一篇 C# 1.0



第 1 章 C#简介

C#（发音为“See Sharp”）是简单、现代、面向对象和类型安全的编程语言。C#起源于 C 语言家族，因此，C、C++和 Java 的程序员很快就能熟悉它。C#已经获得了 ECMA International 和 ISO/IEC 的国际标准认证，它们分别是 *ECMA-334* 标准和 *ISO/IEC 23270* 标准。Microsoft 用于 .NET 框架的 C#编译器就是根据这两个标准实现的。

C#是面向对象的语言，但 C#进一步支持**面向组件（component-oriented）**的编程。现代软件设计越来越依赖于自包含（self-contained）和自描述（self-describing）功能包形式的软件组件。这种组件的关键之处在于它们提供了带有属性、方法和事件的编程模型；它们还具有提供关于组件声明信息的特性（attribute）；同时，它们编入了自己的文档。C#提供语言构件来直接支持这些概念，使得 C#在创建和使用软件构件方面非常自然。

C#的一些特征支持创建健壮和持久的应用程序：**垃圾回收（garbage collection）**特征可以自动回收无用对象所占的内存空间；**异常处理（exception handling）**提供了一个结构化和可扩展的方式，用于错误检测和恢复；语言的**类型安全（type-safe）**设计了防止引用非初始化变量、数组下标越界，以及执行未检查的类型转换等情形的发生。

C#具有统一类型系统。所有的 C#类型，包括诸如 int 和 double 的基本数据类型，都继承于单个根类型 object。因此，所有类型都共享一组通用操作，并且，任何类型的值都能够以一致的方式存储、传递和操作。C#支持用户自定义引用类型和值类型，既允许对象的动态分配，也允许轻量结构的内联存储。

为了保证 C#程序和类库总能以兼容的方式升级，在 C#的设计中十分强调版本控制。许多程序语言不太重视这一点，导致采用那些语言编写的程序常常因为其所依赖的类库的更新而无法工作。C#的设计方面直接受到版本考虑的影响，包括分开的 virtual 和 override 修饰符、方法重载的规则，以及支持对显式接口成员的声明。

本章的其他部分将描述 C#语言的本质特征。尽管后面的章节将更为详细，有时甚至是细致入微地描述这些规则和异常，但是本章力求对整个 C#做一个简单明了的说明。其意图是向读者提供对语言的入门介绍，以便于读者上手编写程序和阅读后面的章节。

1.1 Hello World

学习某种编程语言，通常采用“Hello, World”程序作为起步。下面是 C#版的：

```
using System;
class Hello
{
    static void Main(){
        Console.WriteLine("Hello,World");
    }
}
```


C#源程序文件一般用.cs 作为扩展名。假定“Hello, World”源程序文件被存为 hello.cs, 那么, 使用下面的命令行就能通过 Microsoft C#编译器编译这个程序:

```
csc hello.cs
```

它将产生一个名为 hello.exe 的可执行程序集。当程序运行时, 输出结果如下:

```
Hello,World
```

“Hello, World”程序开头是 using 指令, 引用了 System 命名空间 (namespace)。命名空间提供了 C#程序和类库分层次的组织手段。命名空间包含类型和其他命名空间, 例如, System 命名空间包含若干类型 (如程序中引用的 Console 类), 以及若干其他命名空间 (如 IO 和 Collections)。如果通过 using 指令引用给定命名空间, 就可以对命名空间的成员进行非限定的使用。正是由于程序中使用了 using 指令, 才能够将 System.Console.WriteLine 简写为 Console.WriteLine。

“Hello, World”程序中声明的 Hello 类只有一个成员, 即名为 Main 的方法。Main 方法是用 static 修饰符声明的。静态方法不同于实例方法, 后者需要使用关键字 this 来引用特定的对象实例, 而静态方法的操作不需要引用特定的对象。作为约定, 被命名为 Main 的静态方法充当程序的入口点。

程序输出是由 System 命名空间下 Console 类的 WriteLine 方法产生的。这个类是由 .NET 框架类库提供的, 默认情况下, 类库被 Microsoft C#编译器自动引用。注意 C#本身没有单独的运行时类库。事实上, .NET 框架是 C#的运行时类库。

1.2 程序结构

C#中程序结构的关键概念为**程序**、**命名空间**、**类型**、**成员**和**程序集**。C#程序包括一个或多个源文件。程序中声明类型, 类型包含成员并能够被组织到命名空间中。类和接口是类型的例子。字段、方法、属性和事件则是成员的例子。当 C#程序被编译时, 它们被物理地打包到程序集中。程序集的文件扩展名一般为.exe 或者.dll, 这取决于它们是实现为**应用程序 (application)**, 还是**类库 (library)**。

示例:

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
        public void Push(object data){
            top=new Entry(top,data);
        }
        public object Pop(){
            if (top==null) throw new InvalidOperationException();
            object result=top.data;
        }
    }
}
```

```

        top=top.next;
        return result;
    }
    class Entry
    {
        public Entry next;
        public object data;

        public Entry(Entry next,object data){
            this.next=next;
            this.data=data;
        }
    }
}
}

```

在叫做 `Acme.Collections` 的命名空间下，声明名为 `Stack` 的类，这个类的完全限定名就是 `Acme.Collections.Stack`。它包括几个成员：一个名为 `top` 的字段，两个分别命名为 `push` 和 `pop` 的方法，以及一个名为 `Entry` 的嵌套类。`Entry` 类又进一步包括三个成员：一个名为 `next` 的字段，一个名为 `data` 的字段，以及一个构造函数。假定这个示例的源程序被存为 `acme.cs` 文件，命令行为：

```
csc /t:library acme.cs
```

将这个示例编译为类库（不带 `Main` 入口点的代码），并且产生一个名为 `acme.dll` 的程序集。

程序集包括**中间语言（Intermediate Language, IL）**指令形式的可执行代码，以及**元数据（metadata）**形式的符号信息。在它执行之前，程序集的 IL 代码将被 .NET 公共语言运行库（**Common Language Runtime, CLR**）自动转换成特定处理器的代码。

由于程序集是自描述的功能单元，它既包括代码，也包括元数据，因此，在 C# 中不需要 `#include` 指令和头文件。假如某个 C# 程序需要引用特定程序集中的公共类型和成员，那么只在编译时简单地引用那个程序集就可以了。例如，下面的程序使用来自 `acme.dll` 程序集中的 `Acme.Collections.Stack` 类：

```

using System;
using Acme.Collections;

class Test
{
    static void Main(){
        Stack s=new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}

```

如果程序被存为 `test.cs` 文件，那么，在 `test.cs` 被编译时，`acme.dll` 可以通过 `/r` 选项被引用：

```
rsc /r:acme.dll test.cs
```

这样可以创建一个名为 test.exe 的可执行程序集，运行结果如下：

```
100
10
.
```

C#允许一个程序的源文本被存为几个源文件。当多文件的 C#程序被编译时，所有的源文件都被一起处理，并且各个源文件从概念上能够自由地相互引用，就如同处理之前，所有的源文件被连接成一个大文件。在 C#中向前声明是没有必要的，原因就是声明的顺序无关紧要。C#不限制一个源文件只能声明一个公共类型，也不要求源文件名必须与该文件中的类型相匹配^[注 1]。

1.3 类型和变量

C#中有两种类型：**值类型 (value type)** 和 **引用类型 (reference type)**。值类型变量直接包括它们的数据，而引用类型变量存储的是它们的数据引用，后者被认为是对象。对于引用类型，有可能两个变量引用相同的对象，因此对其中一个变量的操作可能影响另一个对象引用的对象。对于值类型，每个变量都有自己的数据拷贝，因此对一个变量的操作不可能影响其他变量 (ref 和 out 参数变量例外)。

C#的值类型进一步划分为**简单类型 (simple type)**、**枚举类型 (enum type)** 和 **结构类型 (struct type)**；C#的引用类型进一步划分为**类类型 (class type)**、**接口类型 (interface type)**、**数组类型 (array type)** 和 **委托类型 (delegate type)**。

表 1.1 为整个 C#类型系统的概述。

表 1.1 C#类型系统的概述

类 别		描 述
值类型	简单类型	有符号整型: sbyte,short,int,long
		无符号整型: byte,ushort,uint,ulong
		Unicode 字符: char
		IEEE 浮点型: float,double
		高精度小数: decimal
		布尔型: bool
引用类型	枚举类型	用户自定义类型 enum E{...}
	结构类型	用户自定义类型 struct S{...}
	类类型	所有其他类型的最终基类: object
		Unicode 字符串: string
		用户自定义类型 class C{...}
	接口类型	用户自定义类型 interface I{...}
	数组类型	单维与多维数组, 例如, int[]与 int[,]
	委托类型	用户自定义类型 delegate T D(...)

8 个整型类型分别支持 8 位、16 位、32 位和 64 位整数的有符号或者无符号格式。

^[注 1] 这两个限制在 Java 编程语言中是要求的。

两个浮点类型，float 和 double，分别用 32 位单精度和 64 位双精度的 IEEE754 格式表示。

decimal 是 128 位的数据类型，适合财金和货币方面的计算。

C#的 bool 类型用于表示布尔值——true 或者 false。

在 C#中，字符和字符串的处理使用 Unicode 编码。char 类型表示 16 位的 Unicode 编码单元，string 类型表示 16 位的 Unicode 编码单元的序列。

表 1.2 总结了 C#的数值类型。

表 1.2 C#的数值类型

类 别	位 数	类 型	范围 / 精度
有符号整型	8	sbyte	-128 ~ 127
	16	short	-32 768 ~ 32 767
	32	int	-2 147 483 648 ~ 2 147 483 647
	64	long	-9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807
无符号整型	8	byte	0 ~ 255
	16	ushort	0 ~ 65535
	32	uint	0 ~ 4 294 967 295
	64	ulong	0 ~ 18 446 744 073 709 551 615
浮点型	32	float	$1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$ ，7 位精度
	64	double	$5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$ ，15 位精度
Decimal	128	decimal	$1.0 \times 10^{-28} \sim 7.9 \times 10^{28}$ ，28 位精度

C#程序使用类型声明创建新类型。类型声明指定新类型的名字和成员。有 5 种 C# 类型可由用户自定义：类类型、结构类型、接口类型、枚举类型和委托类型。

- 类类型定义了一个数据结构，它包括数据成员（字段）和函数成员（方法、属性及其他）。类类型支持继承和多态，即派生类能够扩展和特殊化基类的机制。
- 结构类型与类类型相似，表示带有数据成员和函数成员的结构。然而，与类类型不同的是，结构是值类型，不需要堆分配。结构不支持用户指定的继承，所有的结构类型隐式地继承类型 object。
- 接口类型定义了一个约定，作为一组函数成员命名的集合。实现接口的类或结构必须提供接口函数成员的实现。接口可能从多个基接口继承而来，类或结构也可能实现多个接口。
- 枚举类型是带有命名常量的独特类型。每个枚举类型有一个底层的类型，它必须是 8 个整型类型之一。枚举类型的值集与底层类型的值集相同。
- 委托类型通过特定的参数列表和返回类型表示对方法的引用。委托将方法处理为实体，实体能够赋值给变量，并且当做参数传递。委托类似于某些程序语言中的方法指针，不同之处在于，委托是面向对象的，并且是类型安全的。

C#支持任何类型的一维和多维数组。不同于其他类型，数组类型在它们被使用前不必声明。事实上，数组类型的构造是由某个类型名加上方括号。例如，int[]是 int 的一维数组，int[,]是 int 的二维数组，而 int[][]是 int 的一维数组的一维数组。

C#的类型系统是统一的，这样任何类型的值都能够被处理成对象。C#中每一个类型直

接或者间接从 object 类继承而来，并且 object 是所有类型最终的基类。值类型的值可以通过执行装箱（boxing）和取消装箱（unboxing）的操作处理为对象。在下面的示例中，int 被转换为 object，然后又转回到 int。

```
using System;

class Test
{
    static void Main()
    {
        int i=123;
        object o=i;    //装箱
        int j=(int)o;   //取消装箱
    }
}
```

当值类型的值被强制类型转换为 object 时，就会分配持有该值的对象实例（也称为“箱子”），并且值也被拷贝到那个箱子里。相反地，当 object 引用被强制类型转换为值类型时，要检查这个引用类型是否是当前值类型的箱子，如果是的话，箱子中的值就会被拷贝出来。

C#统一的类型系统意味着值类型能够“按需”转换为对象。由于这种统一性，使用 object 类型的通用类库，例如.NET 框架中的集合类，能够通过引用类型和值类型使用。

C#中存在几种变量，包括字段、数组元素、局部变量和参数。变量表示了存储的位置，并且每一个变量都有一个类型，以决定什么样的值能够存入变量中，如表 1.3 所示。

表 1.3 C#的变量

变量类型	可能的内容
值类型	值类型的值
object	null 引用、任何引用类型的对象引用或任何值类型装箱值的引用
类类型	null 引用、该类类型实例的引用或由该类类型派生类的实例的引用
接口类型	null 引用、实现该接口的类类型实例的引用或实现该接口的值类型装箱值的引用
数组类型	null 引用、该数组类型实例的引用或兼容数组类型实例的引用
委托类型	null 引用或该委托类型实例的引用

1.4 表达式

表达式（expression）由操作数（operand）和运算符（operator）构成。表达式的运算符标明在操作数上运用了哪种操作。运算符的例子包括+、-、*、/和 new。操作数的例子包括字面值、字段、局部变量和表达式。

当表达式包括多个运算符时，运算符的**优先级（precedence）**控制各个运算符执行的顺序。例如，表达式 x+y*z 将以 x+(y*z)的形式计算，原因就是运算符“*”的优先级高于运算符“+”。

大多数运算符能够被**重载（overload）**。运算符的重载允许用户自定义运算符实现，用于为用户自定义的类或者结构类型指定操作方式。

表 1.4 总结了 C#的运算符，运算符的分类排列是按其优先级从高到低的次序。同一分类的运算符具有相同的优先级。

表 1.4 C#的运算符

分 类	表 达 式	描 述
基本	x.m	成员访问
	x(...)	方法和委托调用
	x[...]	数组和索引器访问
	x++	后增量 (post-increment)
	x--	后减量 (post-decrement)
	new T(...)	对象和委托创建
	new T[...]	数组创建
	typeof(T)	获得 T 类型的 System.Type 对象
	checked(x)	在检查的上下文计算表达式
	unchecked(x)	在未检查的上下文计算表达式
一元	+x	表达式的值相同
	-x	求相反数
	!x	逻辑求反
	~x	按位求反
	++x	前增量 (pre-increment)
	--x	前减量 (pre-decrement)
	(T)x	显式地将 x 的类型转换为类型 T
乘除法	x*y	乘
	x/y	除
	x%y	求余
加减	x+y	加, 字符串合并, 委托组合
	x-y	减, 委托移除
移位	x<<y	左移
	x>>y	右移
关系和类型检测	x<y	小于
	x>y	大于
	x<=y	小于或者等于
	x>=y	大于或者等于
	x is T	如果 x 属于 T 类型, 返回 true; 否则, 返回 false
	x as T	返回转换为类型 T 的 x; 如果 x 不是 T, 就返回 null ^{[1] 2}
相等	x==y	等于
	x!=y	不等于
逻辑与	x&y	整型按位与, 布尔型逻辑与
逻辑异或	x^y	整型按位异或, 布尔型逻辑异或
逻辑或	x y	整型按位或, 布尔型逻辑或
条件与	x&&y	如果 x 为 true, 则计算 y
条件或	x y	如果 x 为 false, 则计算 y
条件	x?y:z	如果 x 为 true, 则计算 y; 如果 x 为 false, 则计算 z
赋值	x=y	赋值
	x op=y	复合赋值; 支持的运算符有: *= /= %= += -= <<= >>= &= ^= =

1.5 语句

程序的活动是通过**语句 (statement)**来表达的。C#支持几种不同的语句, 许多语句是以嵌入语句的形式定义的。

块 (block) 允许在只能使用单个语句的上下文中编写多个语句。块由一个括在大括号

^{[1] 2} as 运算符用于将一个值显式地转换 (使用引用转换或装箱转换) 为一个给定的引用类型; 如果指定的转换不可能实施, 则运算结果为 null。

“{}”内的语句列表组成。

声明语句 (declaration statement) 用于声明局部变量和常量。

表达式语句 (expression statement) 用于运算表达式。表达式可以作为语句使用^{注 3}，包括方法调用、使用 new 运算符进行对象分配、使用 “=” 和复合赋值运算符进行赋值，以及使用 “++” 和 “--” 运算符进行增量和减量的运算。

选择语句 (selection statement) 用于根据某个表达式的值，选择执行若干可能语句中的某一个。这一组语句有 if 和 switch 语句。

迭代语句 (iteration statement) 用于重复执行嵌入语句。这一组语句有 while, do, for 和 foreach 语句。

跳转语句 (jump statement) 用于传递程序控制。这一组语句有 break, continue, goto, throw 和 return 语句。

try-catch 语句用于捕捉在块的执行期间发生的异常。并且，try-finally 语句用于指定一个终止代码块，不管异常出现与否，它总是被执行。

checked 和 unchecked 语句用于控制整型算术运算和转换的溢出检查上、下文。

lock 语句用于获取给定对象的互斥锁，执行语句，然后释放该锁。

using 语句用于获取一个资源，执行一个语句，然后处理该资源。

表 1.5 列出了 C#的语句，并逐个提供了示例。

表 1.5 C#的语句

语 句	示 例
局部变量声明	<pre>static void Main(){ int a; int b=2,c=3; a=1; Console.WriteLine(a+b+c); }</pre>
局部常量声明	<pre>static void Main(){ const float pi=3.1415927f; const int r=25; Console.WriteLine(pi * r * r); }</pre>
表达式语句	<pre>static void Main(){ int i; i=123; //表达式语句 Console.WriteLine(i); //表达式语句 i++; //表达式语句 Console.WriteLine(i); //表达式语句 }</pre>
if 语句	<pre>static void Main(string[] args){ if(args.Length == 0){ Console.WriteLine("No arguments"); } else{ Console.WriteLine("One or more arguments"); } }</pre>

^{注 3} 不是所有的表达式都能作为语句使用。例如，x + y 和 x == 1 只计算一个值（此值将被放弃）的表达式，不能作为语句使用。

(续表)

语 句	示 例
switch 语句	<pre>static void Main(string[] args){ int n = args.Length; switch(n){ case 0: Console.WriteLine("No arguments"); break; case 1: Console.WriteLine("One argument"); break; default: Console.WriteLine("{0} arguments", n); break; } }</pre>
while 语句	<pre>static void Main(string[] args){ int i = 0; while(i < args.Length){ Console.WriteLine(args[i]); i++; } }</pre>
do 语句	<pre>static void Main(){ string s; do{ s = Console.ReadLine(); if(s!=null) Console.WriteLine(s); } while(s != null); }</pre>
for 语句	<pre>static void Main(string[] args){ for(int i = 0; i < args.Length; i++){ Console.WriteLine(args[i]); } }</pre>
foreach 语句	<pre>static void Main(string[] args){ foreach(string s in args){ Console.WriteLine(s); } }</pre>
break 语句	<pre>static void Main(){ while(true){ string s = Console.ReadLine(); if (s == null) break; Console.WriteLine(s); } }</pre>
continue 语句	<pre>static void Main(string[] args){ for(int i = 0; i < args.Length; i++){ if (args[i].StartsWith("/")) continue; Console.WriteLine(args[i]); } }</pre>
goto 语句	<pre>static void Main(string[] args){ int i = 0; goto check; loop: Console.WriteLine(args[i++]); check: if (i < args.Length) goto loop; }</pre>

(续表)

语 句	示 例
return 语句	<pre>static int Add(int a, int b){ return a + b; } static void Main(){ Console.WriteLine(Add(1, 2)); return; }</pre>
throw 和 try 语句	<pre>static double Divide(double x, double y) { if (y == 0) throw new DivideByZeroException(); return x / y; } static void Main(string[] args){ try{ if (args.Length !=2){ throw new Exception("Two numbers required"); } double x = double.Parse(args[0]); double y = double.Parse(args[1]); Console.WriteLine(Divide(x, y)); } catch (Exception e) { Console.WriteLine(e.Message); } }</pre>
checked 和 unchecked 语句	<pre>static void Main(){ int i = int.MaxValue; checked { Console.WriteLine(i + 1); //异常 } unchecked { Console.WriteLine(i + 1); //溢出 } }</pre>
lock 语句	<pre>class Account { decimal balance; public void Withdraw(decimal amount) { lock(this) { if (amount > balance) { throw new Exception("Insufficient funds"); } balance -= amount; } } }</pre>
using 语句	<pre>static void Main(){ using (TextWriter w = File.CreateText("test.txt")) { w.WriteLine("Line one"); w.WriteLine("Line two"); w.WriteLine("Line three"); } }</pre>

1.6 类和对象

类（class）是 C# 类型中最基础的类型。类是一个数据结构，将状态（字段）和行为（方法和其他函数成员）组合在一个单元中。类提供了用于动态创建类实例的定义，也就是对象（object）。类支持继承（inheritance）和多态（polymorphism），即派生类能够扩展和

特殊化基类的机制。

使用类声明可以创建新的类。类声明以一个声明头开始，其组成方式如下：先是指定类的特性和修饰符，后跟类的名字，基类（如果有的话）的名字，以及被该类实现的接口名。声明头后面就是类体了，它由一组包含在大括号（{ }）中的成员声明组成。

下面是一个名为 Point 的简单类的声明：

```
public class Point
{
    public int x, y;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

使用 new 运算符创建类的实例，它将为新实例分配内存，调用构造函数初始化实例，并且返回对该实例的引用。下面的语句创建两个 Point 对象，并且将那些对象的引用保存到两个变量中：

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

当不再使用对象时，该对象所占的内存将被自动回收。在 C#中，没有必要也不可能显式地释放对象。

1.6.1 成员

类的成员或者是静态成员（static member），或者是实例成员（instance member）。静态成员属于类，实例成员属于对象（类的实例）。

表 1.6 提供了类所能包含的各种成员的描述。

表 1.6 类的成员

成 员	描 述
常数	与类关联的常量值
字段	类的变量
方法	能够被类执行的计算和行为
属性	使对象能够读取和写入类的命名属性
索引器	使对象能够用与数组相同的方式进行索引
事件	能够被类产生的通知
运算符	类支持的转换和表达式运算符
构造函数	初始化类的实例或者类本身
析构函数	在永久销毁类的实例之前执行的行为
类型	被类声明的嵌套类型

1.6.2 可访问性

类的每个成员都有关联的可访问性，它控制能够访问该成员的程序文本区域。有 5 种可能的可访问性形式。表 1.7 概述了类的可访问性的意义。

表 1.7 类的可访问性

可访问性	意 义
public	访问不受限制
protected	访问仅限于包含类或从包含类派生的类型
internal	访问仅限于当前程序集
protected internal	访问仅限于从包含类派生的当前程序集或类型
private	访问仅限于包含类

1.6.3 基类

类的声明可能通过在类名后加上冒号和基类的名字来指定一个基类^{译注 4}。省略基类等同于直接从 object 类派生。在下面的示例中，Point3D 的基类是 Point，而 Point 的基类是 object：

```
public class Point
{
    public int x, y;
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
public class Point3D: Point
{
    public int z;
    public Point3D(int x, int y, int z): Point(x, y){
        this.z = z;
    }
}
```

Point3D 类继承了其基类的成员。继承意味着类将隐式地包含其基类的所有成员（除了基类的构造函数）。派生类能够在继承基类的基础上增加新的成员，但是它不能移除继承成员的定义。在前面的示例中，Point3D 类从 Point 类中继承了 x 字段和 y 字段，并且每一个 Point3D 实例都包含三个字段 x，y 和 z。

从类类型到它的任何基类类型都存在隐式的转换。并且，类类型的变量能够引用该类的实例，或者任何派生类的实例。例如，对于前面给定的类声明，Point 类型的变量能够引用 Point 实例或者 Point3D 实例：

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

^{译注 4} 基类（base class）对应 Java 编程语言中的超类（或者父类；super class）。

1.6.4 字段

字段是与对象或类相关联的变量。

当一个字段声明中含有 `static` 修饰符时，由该声明引入的字段为**静态字段 (static field)**。它只标识了一个存储位置。不管创建了多少个类实例，静态字段都只会会有一个副本。

当一个字段声明中不含有 `static` 修饰符时，由该声明引入的字段为**实例字段 (instance field)**。类的每个实例都包含了该类的所有实例字段的一个单独副本。

在下面的示例中，`Color` 类的每个实例都有 `r`, `g`, `b` 实例字段的不同副本，但是 `Black`, `White`, `Red`, `Green` 和 `Blue` 等静态字段只有一个副本：

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

如前面的示例所示，通过 `readonly` 修饰符声明只读字段。给 `readonly` 字段的赋值只能作为声明的组成部分出现，或者在同一类中的实例构造函数或静态构造函数中出现。

1.6.5 方法

方法 (method) 是一种用于实现可以由对象或类执行的计算或操作的成员。**静态方法 (static method)** 只能通过类来访问。**实例方法 (instance method)** 则要通过类的实例访问。

方法有一个**参数 (parameter)** 列表 (可能为空)，表示传递给方法的值或者引用；方法还有**返回类型 (return type)**，用于指定由该方法计算和返回的值的类型。如果方法不返回一个值，则它的返回类型为 `void`。

在声明方法的类中，该方法的签名必须是惟一的。方法的签名由它的名称、参数的数目、每个参数的修饰符和类型组成。返回类型不是方法签名的组成部分。

1.6.5.1 参数

参数用于将值或者引用变量传递给方法。当方法被调用时，方法的参数^{译注 5}从指定的

^{译注 5} 相当于形式参数 (简为形参)。

自变量 (argument)^[6] 得到它们实际的值。C#有4种参数：值参数、引用参数、输出参数和参数数组。

值参数 (value parameter) 用于输入参数的传递。值参数相当于一个局部变量，它的初始值是从为该参数所传递的自变量获得的。对值参数的修改不会影响所传递的自变量。

引用参数 (reference parameter) 用于输入和输出参数的传递。用于引用参数的自变量必须是一个变量，并且在方法执行期间，引用参数和作为自变量的变量所表示的是同一个存储位置。引用参数用 `ref` 修饰符声明。下面的示例展示了 `ref` 参数的使用：

```
using System;
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j); //输出 "2 1"
    }
}
```

输出参数 (output parameter) 用于输出参数的传递。输出参数类似于引用参数，不同之处在于调用方提供的自变量初始值无关紧要。输出参数用 `out` 修饰符声明。下面的示例展示了 `out` 参数的使用：

```
using System;
class Test {
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }
    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem); //输出 "3 1"
    }
}
```

参数数组 (parameter array) 允许将可变长度的自变量列表传递给方法。参数数组用 `params` 修饰符声明。只有方法的最后一个参数能够被声明为参数数组，而且它必须是一维数组类型。`System.Console` 类的 `Write` 和 `WriteLine` 方法是参数数组应用的很好的例子。它们的声明形式如下：

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}

    public static void WriteLine(string fmt, params object[] args) {...}
}
```

^[6] 相当于实际参数（简为实参）。

```
...
}
```

在方法中使用参数数组时，参数数组表现得就像常规的数组类型参数一样。然而，带数组参数的方法调用中，既可以传递参数数组类型的单个自变量，也可以传递参数数组的元素类型的若干自变量。对于后者的情形，数组实例将自动被创建，并且通过给定的自变量初始化。示例：

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

等价于下面的语句：

```
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine("x={0} y={1} z={2}", args);
```

1.6.5.2 方法体和局部变量

方法体指定方法调用时所执行的语句。

方法体能够声明特定于该方法调用的变量。这样的变量被称为**局部变量 (local variable)**。局部变量声明指定类型名、变量名，可能还有初始值。下面的示例声明了一个局部变量 *i*，其初始值为 0；另一个局部变量 *j* 没有初始值。

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while(i < 10){
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C#要求局部变量在其值被获得之前**明确赋值 (definitely)**。例如，假设前面的变量 *i* 的声明没有包含初始值，那么，在接下来对 *i* 的使用将导致编译器报告错误，原因就是 *i* 在程序中没有明确赋值。

方法能够使用 **return** 语句将控制返回给它的调用方。如果方法是 **void** 的，则 **return** 语句不能指定表达式；如果方法是非 **void** 的，则 **return** 语句必须包含表达式，用于计算返回值。

1.6.5.3 静态方法和实例方法

若一个方法声明中含有 **static** 修饰符，则称该方法为**静态方法 (static method)**。静态方法不对特定实例进行操作，只能访问静态成员。

若一个方法声明中没有 **static** 修饰符，则称该方法为**实例方法 (instance method)**。实

例方法对特定实例进行操作，既能够访问静态成员，也能够访问实例成员。在调用实例方法的实例上，可以用 `this` 来访问该实例，而在静态方法中引用 `this` 是错误的。

下面的 `Entity` 类具有静态和实例两种成员：

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

每一个 `Entity` 实例包含一个序列号（并且假定这里省略了一些其他信息）。`Entity` 构造函数（类似于实例方法）用下一个有效的序列号初始化新的实例。因为构造函数是一个实例成员，所以，它既可以访问 `serialNo` 实例字段，也可以访问 `nextSerialNo` 静态字段。

`GetNextSerialNo` 和 `SetNextSerialNo` 静态方法能够访问 `nextSerialNo` 静态字段，但是如果访问 `serialNo` 实例字段就会产生错误。

下面的示例展示了 `Entity` 类的使用：

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();

        Console.WriteLine(e1.GetSerialNo());           //输出 "1000"
        Console.WriteLine(e2.GetSerialNo());           //输出 "1001"
        Console.WriteLine(Entity.GetNextSerialNo());    //输出 "1002"
    }
}
```

注意，`SetNextSerialNo` 和 `GetNextSerialNo` 静态方法通过类调用，而 `GetSerialNo` 实例成员则通过类的实例调用。

1.6.5.4 虚拟方法、重写方法和抽象方法

若一个实例方法的声明中含有 `virtual` 修饰符，则称该方法为**虚拟方法**（`virtual method`）。若其中没有 `virtual` 修饰符，则称该方法为**非虚拟方法**（`nonvirtual method`）。

在一个虚拟方法调用中，该调用所涉及的实例的**运行时类型 (runtime type)** 确定了要被调用的究竟是该方法的哪一个实现。在非虚拟方法调用中，实例的**编译时类型 (compile-time type)** 是决定性因素。

虚拟方法可以由派生类**重写 (override)**^{译注7}实现。当一个实例方法声明中含有 `override` 修饰符时，该方法将重写所继承的相同签名的虚拟方法。虚拟方法声明用于引入新方法，而重写方法声明则用于使现有的继承虚拟方法专用化（通过提供该方法的新实现）。

抽象 (abstract) 方法是没有实现的虚拟方法。抽象方法的声明是通过 `abstract` 修饰符实现的，并且只允许在抽象类中使用抽象方法声明。非抽象类的派生类需要重写抽象方法。

下面的示例声明了一个抽象类 `Expression`，它表示一个表达式树的节点；它有三个派生类 `Constant`，`VariableReference`，`Operation`，它们实现了常数、变量引用和算术运算的表达式树节点。

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;
    public Constant(double value) {
        this.value = value;
    }
    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;
    public VariableReference(string name) {
        this.name = name;
    }
    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;
```

^{译注7} `override` 在 Java 世界中常常译为“覆盖”；这里，根据 MSDN, Microsoft .NET Visual Studio 2002/2003 的帮助，将其译为“重写”。

```

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }
    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch(op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}

```

前面的 4 个类用于模型化算术表达式。例如，使用这些类的实例，表达式 $x+3$ 能够被表示为如下的形式：

```

Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));

```

Expression 实例的 **Evaluate** 方法将被调用，以计算表达式的值，从而产生一个 **double** 值。该方法取得一个包含变量名（输入的键）和值（输入的值）的 **Hashtable** 作为其自变量。**Evaluate** 方法是虚拟的抽象方法，意味着派生类必须重写它并提供实际的实现。

Evaluate 方法的 **Constant** 的实现只是返回保存的常数。**VariableReference** 的实现是在 **Hashtable** 中查找变量名，并且返回相应的值。**Operation** 的实现则首先计算左操作数和右操作数的值（通过递归调用 **Evaluate** 方法），然后执行给定的算术运算。

下面的程序使用 **Expression** 类，对于不同的 x 和 y 的值，计算表达式 $x*(y+2)$ 。

```

using System;
using System.Collections;
class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        Vars["x"] = 3;
        Vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars)); //输出 "21"
        Vars["x"] = 1.5;
        Vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars)); //输出 "16.5"
    }
}

```


1.6.5.5 方法重载

方法重载 (Method overloading) 允许在同一个类中采用同一个名称声明多个方法，条件是它们的签名是惟一的。当编译一个重载方法的调用时，编译器采用**重载决策 (overload resolution)** 确定应调用的方法。重载决策找到最佳匹配自变量的方法，或者在没有找到最佳匹配的方法时报告错误信息。下面的示例展示了重载决策工作机制。在 Main 方法中每一个调用的注释说明了实际被调用的方法。

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }
    static void F(object x) {
        Console.WriteLine("F(object)");
    }
    static void F(int x) {
        Console.WriteLine("F(int)");
    }
    static void F(double x) {
        Console.WriteLine("F(double)");
    }
    static void F(double x, dpuble y) {
        Console.WriteLine("F(double, double)");
    }
    static void Main(){
        F();           //调用 F()
        F(1);          //调用 F(int)
        F(1.0);         //调用 F(double)
        F("abc");       //调用 F(object)
        F((double)1);    //调用 F(double)
        F((object)1);    //调用 F(object)
        F(1, 1);        //调用 F(double, double)
    }
}
```

如上例所示，总是通过自变量到参数类型的显式的类型转换，来选择特定方法。

1.6.6 其他函数成员

类的函数成员 (function member) 是包含可执行语句的成员。前面部分所描述的方法是主要的函数成员。这一节讨论其他几种 C#支持的函数成员：构造函数、属性、索引器、事件、运算符、析构函数。

表 1.8 展示一个名为 List 的类，它实现一个可扩展的对象列表。这个类包含了最通用的几种函数成员的例子。

表 1.8 类的函数成员示例

public class List	
{	
const int defaultCapacity = 4;	常数
object[] items; int count;	字段

(续表)

<pre>public List(): this(defaultCapacity) {} public List(int capacity) { items = new object[capacity]; }</pre>	构造函数
<pre>public int Count { get { return count; } } public string Capacity { get { return items.Length; } set { if (value < count) value = count; if (value != items.Length) { object[] newItems = new object[value]; Array.Copy(items, 0, newItems, 0, count); items = newItems; } } }</pre>	属性
<pre>public object this[int index] { get { return items[index]; } set { items[index] = value; OnListChange(); } }</pre>	索引器
<pre>public void Add(object item) { if (count == Capacity) Capacity = count * 2; items[count] = item; count++; OnChanged(); } protected virtual void OnChanged() { if (Changed != null) Changed(this, EventArgs.Empty); } public override bool Equals(object other) { return Equals (this,other as List); } static bool Equals (List a,List b) { if (a == null) return b == null; if (b == null a.count != b.count) return false; for (int i = 0; i < a.count; i++) { if (!object.Equals(a.item[i], b.item[i])) { return false; } } }</pre>	方法
<pre>public event EventHandler Changed;</pre>	事件
<pre>public static bool operator ==(List a, List b) { return Equals(a, b); } public static bool operator !=(List a, List b) { return !Equals(a, b); }</pre>	运算符
<pre>}</pre>	

1.6.6.1 构造函数

C#既支持实例构造函数，也支持静态构造函数。**实例构造函数 (instance constructor)**是实现初始化类实例所需操作的成员。**静态构造函数 (static constructor)**是一种在类首次加载时用于实现初始化类本身所需操作的成员。

构造函数的声明如同方法一样，不过，它没有返回类型，它的名字与包含它的类名一样。若构造函数的声明中包含 `static` 修饰符，则它声明了一个静态构造函数，否则声明实例构造函数。

实例构造函数能够被重载。例如，`List` 声明了两个实例构造函数，一个不带参数，一个带有一个 `int` 参数。使用 `new` 运算符可以调用实例参数。下面的语句使用各个 `List` 类的构造函数创建了两个 `List` 实例。

```
List list1 = new List();
List list2 = new List(10);
```

实例构造函数不同于其他方法，它是不能被继承的。并且，一个类除了自己声明的实例构造函数外，不可能有其他的实例构造函数。如果一个类没有声明任何实例构造函数，则会自动地为它提供一个默认的空的实例构造函数。

1.6.6.2 属性

属性 (property) 是字段的自然扩展，两者都是具有关联类型的命名成员，而且访问字段和属性的语法是相同的。然而，属性与字段不同，不表示存储位置。相反，属性有**访问器 (accessor)**，这些访问器指定在它们的值被读取或写入时需执行的语句。

属性的声明类似于字段，不同之处在于属性的声明以定界符 `{}` 之间的 `get` 访问器和 / 或 `set` 访问器结束，而不是分号。同时包含 `get` 访问器和 `set` 访问器的属性称为**读写属性 (read-write property)**。只具有 `get` 访问器的属性称为**只读属性 (read-only property)**。只具有 `set` 访问器的属性称为**只写属性 (write-only property)**。

`get` 访问器相当于一个具有属性类型返回值的无参数方法。除了作为赋值的目标外，当在表达式中引用属性时，会调用该属性的 `get` 访问器以计算该属性的值。

`set` 访问器相当于一个具有单个名为 `value` 的参数和无返回类型的方法。当一个属性作为赋值的目标，或者作为 `++` 或 `--` 运算符的操作数被引用时，就会调用 `set` 访问器，所传递的自变量将提供新值。

`List` 类声明了两个属性 `Count` 和 `Capacity`，依次是只读和只写的。下面是使用这些属性的示例：

```
List names = new List();
names.Capacity = 100;           //调用 set 访问器
int i = names.Count;           //调用 get 访问器
int j = names.Capacity;        //调用 get 访问器
```

与字段和方法类似，对于实例属性和静态属性，C#两者都支持。静态属性是声明中具有 `static` 修饰符，而实例属性则没有。

属性的访问器可以是虚拟的。当属性声明中包含 `virtual`，`abstract`，`override` 修饰符时，它们将运用到属性访问器。

1.6.6.3 索引器

索引器是这样一个成员：它使对象能够用与数组相同的方式进行索引。索引器的声明与属性很相似，不同之处在于成员的名字是 **this**，后面的参数列表是在定界符（[]）之间。参数在索引器的访问器中是可用的。与属性类似，索引器可以是读写、只读、只写的，并且索引器的访问器也可以是虚拟的。

List 类声明了单个读写索引器，接受一个 **int** 型的参数。通过索引器就可能用 **int** 值索引 List 实例。例如：

```
List names = new List();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = (string) names[i];
    names[i] = s.ToUpper();
}
```

索引器能够被重载，意味着可以声明多个索引器，只要它们的参数个数或类型不同。

1.6.6.4 事件

事件是使对象或类能够提供通知的成员。事件的声明与字段的类似，不同之处在于事件声明包含一个 **event** 关键字，并且事件声明的类型必须是委托类型。

在包含事件声明的类中，事件可以像委托类型的字段一样使用（这样的事件不能是 **abstract**，而且不能声明访问器）。该字段保存了一个委托的引用，表示事件处理程序已经被添加到事件上。如果尚未添加任何事件处理程序，则该字段为 **null**。

List 类声明了名为 **Changed** 的单个事件成员，**Changed** 事件表明有一个新项添加到事件处理程序列表，它由 **OnChanged** 虚拟方法引发，它首先检查事件是否为 **null**（意思是没事件处理程序）。引发事件的通知正好等价于调用事件所表示的委托——因此，不需要特殊的语言构件引发事件。

客户通过**事件处理程序（event handler）**响应事件。使用“+=”运算符添加或者使用“-=”移除事件处理程序。下面的示例添加一个事件处理程序到 List 类的 **Changed** 事件：

```
using System;

class Test
{
    static int changeCount;
    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }
    static void Main() {
        List names = new List();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount); //输出 "3"
    }
}
```

对于要求控制事件的底层存储的更高级场景^{译注 8}，事件的声明可以显式地提供 add 和 remove 访问器，它们在某种程度上类似于属性的 set 访问器。

1.6.6.5 运算符

运算符 (operator) 是一种函数成员，用来定义可应用于类实例的特定表达式运算符的含义。有三种运算符能够被定义：一元运算符、二元运算符和转换运算符。所有的运算符必须声明为 public 和 static。

List 类声明了两个运算符，运算符 “==” 和运算符 “!=”，并且向表达式赋予新的含义，而这些表达式将这些运算符应用到 List 实例上。特别指出，这些运算符定义了两个 List 对象的相等比较，即使用它们的 Equals 方法进行比较。下面的示例使用 “==” 运算符比较两个 List 实例。

```
using System;
class Test
{
    static void Main() {
        List a = new List();
        a.Add(1);
        a.Add(2);
        List b = new List();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b); //输出 "True"
        b.Add(3);
        Console.WriteLine(a == b); //输出 "False"
    }
}
```

第一个 Console.WriteLine 输出 True，原因是两个 List 集合对象包含个数和值都相同的对象。假如 List 没有定义运算符 “==”，那么第一个 Console.WriteLine 将输出 False，因为 a 和 b 引用不同的 List 实例。

1.6.6.6 析构函数

析构函数 (destructor) 是用于实现析构类实例所需操作的成员。析构函数不能带参数，不能具有可访问性修饰符，也不能被显式地调用。垃圾回收期间会自动调用所涉及实例的析构函数。

垃圾回收器在决定何时回收对象和运行析构函数方面采取宽松的策略。特别指出，析构函数的调用时机是不确定的，并且析构函数可能运行在任何线程上。由于这些或者其他原因，只有没有其他可行的解决方案，类才实现析构函数。

1.7 结构

像类一样，**结构 (struct)** 也是表示可以包含数据成员和函数成员的数据结构。但是，

^{译注 8} 例如，为每个事件设置一个字段所造成的内存开销，有时会变得不可接受。在这种情况下，可以在类中使用事件访问器声明，并采用专用机制来存储事件处理程序列表。

与类不同的是，结构是一种值类型，并且不需要堆分配。结构类型的变量直接保存结构的数据，而类类型的变量保存对动态分配对象的引用。结构类型不支持用户指定的继承，并且所有的结构类型隐式地继承于 `object` 类型。

结构对于具有值语义（value semantics）的小的数据结构特别有用。复数、坐标系中的点或字典中的“键—值”对都是结构的典型示例。对于小的数据结构，使用结构而不是类能够节省大量的分配内存。例如，下面的程序创建并初始化 100 点的数组。如果用类来实现 `Point`，则要初始化 101 个单独的对象——数组需要一个，它的 100 个元素每个都需要一个。

```
class Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

将 `Point` 改为作为结构实现：

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

现在，只出现一个实例对象，即用于数组的对象。`Point` 实例在数组中内联存储。

结构构造函数是通过 `new` 运算符调用的，但这并不意味着分配内存。结构构造函数不是动态分配对象并返回引用，而是简单地返回结构值本身（一般在堆栈上的临时位置），这个值在需要时将被拷贝。

对于类，两个变量可能引用同一对象，因此对一个变量进行的操作可能影响另一个变量所引用的对象。对于结构，每个变量都有它们自己的数据副本^{译注 9}，对一个变量的操作不可能影响其他变量。例如，下面的代码片段产生的输出取决于 `Point` 是类还是结构：

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

如果 `Point` 是类，则输出是 20，原因是 `a` 和 `b` 引用的是同一个对象。如果 `Point` 是结

^{译注 9} 除 `ref` 和 `out` 参数变量外。

构，则输出是 10，因为 a 对 b 的赋值，实际上创建了值的副本，因此，接下来对 a.x 的赋值将不会影响到这个副本。

前面的示例强调了结构的两个限制。首先，拷贝整个结构往往比拷贝一个对象引用要低效，因此，将结构用于赋值和值参数传递比引用类型开销大。其次，除了 ref 和 out 参数变量，不可能创建对结构的引用，这样限制了结构的应用范围。

1.8 数组

数组 (array) 是一种包含若干变量的数据结构，这些变量都可以通过计算索引进行访问。数组中包含的变量也称为数组的元素，它们具有相同的类型，该类型称为数组的元素类型。

数组类型是引用类型，并且任何数组变量的声明只是预留空间，用于对数组实例的引用。实际的数组实例是通过 new 运算符在运行时动态创建的。new 运算符指定新的数组实例的**长度 (length)**，它将在实例的生存期间固定下来。数组元素的索引范围从 0 到 Length-1。new 运算符自动将数组的元素初始化为它们的默认值，例如，所有数值类型的数组元素将被初始化为零，而所有引用类型的则被初始化为 null。

下面的示例创建一个 int 型元素的数组，接着初始化该数组，最后将数组的元素打印出来：

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) a[i] = i * i;
        for (int i = 0; i < a.Length; i++){
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

这个示例创建和操作**一维数组 (single-dimensional array)**。C# 也支持**多维数组 (multi-dimensional array)**。数组类型的维数也被认为是数组类型的**秩 (rank)**，它是数组类型的方括号之间逗号的个数加上 1。下面的示例分配了一个一维数组、一个二维数组和一个三维数组：

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

这里，a1 数组包含 10 个元素，a2 数组包含 50(10×5)个元素，a3 数组包含 100(10×5×2)个元素。

数组的元素类型可能是任何类型，包括数组类型。数组的元素为数组有时也称做**交错型数组 (jagged array)**，原因是各个元素数组的长度并不是一致的。下面的示例分配了一

个 int 数组的数组：

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

第一行创建了一个带有三个元素的数组，每个元素的类型是 `int[]`，并且初始值为 `null`。接下来的代码行用不同长度的数组实例的引用初始化这三个元素。

`new` 运算符允许使用**数组初始值设定项**（**array initializer**）指定数组元素的初始值，它实际是定界符 `{}` 之间的表达式列表。下面的示例用三个元素分配和初始化 `int[]`。

```
int[] a = new int[] {1, 2, 3};
```

注意从定界符“`{}`”之间的表达式数目可以推断出数组的长度。对于局部变量和字段的声明，允许使用简写形式，因此没有必要再次声明数组类型。

```
int[] a = {1, 2, 3};
```

前面的两个示例相当于下面的语句：

```
int[] a = new int[3];
a[0] = 1;
a[1] = 2;
a[2] = 3;
```

1.9 接口

接口定义一个约定，它能够被类或结构实现。接口可以包含方法、属性、索引器和事件作为成员。接口不提供它所定义的成员的实现——接口只指定实现该接口的类或结构必须提供的成员。

接口支持多重继承（**multiple inheritance**）。在下面的示例中，接口 `IComboBox` 同时继承于接口 `ITextBox` 和接口 `IListBox`：

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

类和结构能够实现多个接口。在下面的示例中，`EditBox` 类同时实现 `IControl` 接口和 `IDataBound` 接口。

```
interface IDataBound
{
    void Bind(Binder b);
}
```

```

    }

    public class TextBox: IControl, IDataBound
    {
        public void Paint() {...}
        public void Bind(Binder b) {...}
    }

```

当一个类或者结构实现了一个特定的接口，该类或结构能够被隐式地转换为该接口类型。例如：

```

    TextBox textBox = new TextBox();
    IControl control = textBox;
    IDataBound dataBound = textBox;

```

如果接口没有被静态地认为实现了某一个特定的接口，则可以使用动态的类型转换。例如，下面的语句使用动态的类型转换来获得对象的 `IDataBound` 和 `IControl` 的实现。由于对象的实际类型是 `TextBox`，因而类型转换是成功的。

```

    object obj = new TextBox();
    IControl control = (IControl)obj;
    IDataBound dataBound = (IDataBound)obj;

```

在前面的 `TextBox` 类中，使用 `public` 成员分别实现 `IControl` 接口的 `Paint` 方法和 `IDataBound` 接口的 `Bind` 方法。C#也支持**显式接口成员实现**（**explicit interface member implementation**），使得实现接口的类或者结构避免将这些成员设置成 `public`。显式接口成员实现采用完全限定的接口成员名。例如，在 `TextBox` 类中将 `Paint` 方法命名为 `IControl.Paint`，将 `Bind` 方法命名为 `IDataBound.Bind` 方法。

```

    public class TextBox: IControl, IDataBound
    {
        void IControl.Paint() {...}
        void IDataBound.Bind(Binder b) {...}
    }

```

显式接口成员只能通过接口类型来访问。例如，在前面 `TextBox` 类中提供的 `IControl.Paint` 实现，只能通过将对 `TextBox` 的引用强制转换为对 `IControl` 接口类型的引用来调用。

```

    TextBox textBox = new TextBox();
    textBox.Paint();           //错误：没有这样的方法
    IControl control = textBox;
    control.Paint();           //调用 TextBox 的 Paint 实现

```

1.10 枚举

枚举类型（**enum type**）是一组命名常数的值类型。下面的示例声明并使用了一个名为 `Color` 的枚举类型，它含有三个命名常数：`Red`，`Green` 和 `Blue`。

```

    Using system
    enum Color
    {

```

```

        Red,
        Green,
        Blue,
    };
    class Test
    {
        static void PrintColor(Color color) {
            switch(color) {
                case Color.Red:
                    Console.WriteLine("Red");
                    break;
                case Color.Green:
                    Console.WriteLine("Green");
                    break;
                case Color.Blue:
                    Console.WriteLine("Blue");
                    break;
                default:
                    Console.WriteLine("Unknown color");
                    break;
            }
        }
        static void Main() {
            Color c = Color.Red;
            PrintColor(c);
            PrintColor(Color.Blue);
        }
    }
}

```

每个枚举类型都有一个对应的整数类型，称为该枚举类型的**基础类型**（**underlying type**）。没有显式地声明基础类型的枚举声明意味着所对应的基础类型是 `int`。枚举类型的值域不受它的枚举成员限制。具体说来，枚举的基础类型的任何一个值都可以被强制转换为该枚举类型，成为该枚举类型的一个独特的有效值。

下面的示例声明一个名为 `Alignment` 的枚举类型，其基础类型为 `sbyte`。

```

enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}

```

如上例所示，枚举成员的声明中包含常数表达式，用于指定成员的初始值设定项。每个枚举成员的常数值必须在该枚举的基础类型的范围之内。如果枚举成员的声明没有显式地指定初始值，则它被指定为零（如果它是在枚举类型中的第一个成员），或者将前一个枚举成员（按照文本顺序）的关联值加 1。

通过类型强制转换，枚举值能够被转换为整型，反之亦然。例如：

```

int i = (int)Color.Blue;    //int i = 2;
Color c = (Color)2;        //Color c = Color.Blue;

```

任何枚举类型的默认值是转换为该枚举类型的的整型值零。如果变量被自动初始化为默认值，那么，该值就被赋予枚举类型变量。为了使枚举类型的默认值很容易可用，字面零能够隐式地转换为任何枚举类型。因此，下面的语句是允许的：

```
Color c = 0;
```

1.11 委托

委托类型 (delegate type) 用来表示对具有特定参数列表和返回类型的方法的引用。委托将方法处理为实体，使其能够被赋值给变量，并作为参数传递。委托类似于某些其他语言的函数指针的概念。但是，与函数指针不同，委托是面向对象和类型安全的。

下面的示例声明和使用了名为 `Function` 的委托类型。

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;
    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, new Function(Square));
        double[] sines = Apply(a, new Function(Math.Sin));
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, new Function(m.Multiply));
    }
}
```

`Function` 委托类型的实例能够引用任何以 `double` 为自变量并返回 `double` 值的方法。`Apply` 方法将 `Function` 作用于 `double[]` 的元素上，并返回包含结果的 `double[]`。在 `Main` 方法中，通过调用 `Apply` 将三个不同的函数作用于 `double[]`。

委托类型既能够引用静态类型（例如，上例中的 `Square` 或者 `Math.Sin`），也能够引用实例类型（例如，上例中的 `m.Multiply`）。引用实例方法的委托也就引用了一个具体的对象，当通过该委托调用这个实例方法时，在调用中那个对象就成了 `this` 对象。

委托的一个有趣且有用的属性是：它不知道也不关心它所封装的方法所属的类；它所关心的仅限于这些方法的参数和返回类型必须与该委托相同。

1.12 特性

C# 程序中的类型、成员和其他实体支持控制其行为的修饰符。例如，方法的可访问性是通过用修饰符 `public`, `protected`, `internal` 和 `private` 来指定的。C#使程序员可以将这种自定义的声明信息附加到程序实体，而且可以在运行时环境中检索到它们。这种附加的声明信息是通过**特性 (attribute)** 指定的。

下面的示例声明了一个名为 `HelpAttribute` 的特性，该特性可以放在程序实体上，以提供从这些程序实体到其文档说明的映射。

```
using System;
public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }
    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

所有的特性类都是从.NET 框架提供的 `System.Attribute` 基类派生的类。如果特性类的名称以 `Attribute` 为后缀，那么，当引用该特性时可以省略此后缀。例如，可以用如下方式使用 `HelpAttribute` 特性。

```
[HelpAttribute*10 ("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [HelpAttribute ("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

这个示例将一个 `HelpAttribute` 附加到 `Widget` 类上，并且将另一个 `HelpAttribute` 附加到 `Display` 方法上。当特性被附加到程序实体上时，该特性类的公共构造函数控制必须提供的信息。额外的信息则可以通过引用特性类的公共的读/写属性的方法提供（例如，上例中对 `Topic` 的引用）。

下面的示例展示了如何通过反射获取给定程序实体上的特性信息：

```
using System;
using System.Reflection;

class Test
```

^{*10} 原文为 `Help`，应该是作者的笔误。

```
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}: ", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }
    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

当通过反射请求具体的特性时，该特性类的构造函数将被调用，自变量则是在程序源码中提供的，并且返回生成的特性实例。如果通过属性提供特性的额外信息，那么，就在特性实例返回之前将那些属性设置成给定的值。

第 2 章 词法结构

2.1 程序

C#程序由一个或多个**源文件**（**source file**）组成。源文件形式上被称为**编译单元**（**compilation unit**）（§9.1）。源文件是 Unicode 字符的有序序列，它通常与文件系统中的文件有一一对应的关系，但这种对应不是必需的。为了获得最大的可移植性，我们推荐将文件系统中的文件按照 UTF-8 编码方式进行编码。

从概念上讲，程序的编译分三个步骤：

- （1）转换，这一步将用特定字符指令系统和编码方案编写的文件转换为 Unicode 字符序列；
- （2）词法分析，这一步将 Unicode 输入字符流转换为标记流；
- （3）句法分析，这一步将标记流转换为可执行代码。

2.2 文法

本规范采用两种文法(原理)来表示 C# 编程语言的句法。词法文法 (Lexical Grammar) (§ 2.2.2) 规定怎样将 Unicode 字符结合起来构成行结束符、空白、注释、标记和预处理指令等。句法文法 (Syntactic Grammar) (§ 2.2.3) 规定如何把那些由词法文法产生的标记再结合起来构成 C# 程序。

2.2.1 文法表示法

词法文法和句法文法是使用**文法产生式**（**grammar production**）来表示的。每个文法产生式定义一个非结束符号和它可能的扩展（由非结束符或结束符组成的序列）。在文法产生式中，非结束符号显示为斜体，而结束符号显示为等宽字体。

文法产生式的第一行是该产生式所定义的非结束符号的名称后跟一个冒号。每个后续的缩进行包含一个可能的扩展，它是以非结束符或结束符组成的序列的形式给出的。例如，产生式

while-statement: (**while** 语句:)

while (**boolean-expression**) **embedded-statement** (嵌入语句)

定义了一个 **while** 语句。它是这样构成的：由标记 **while** 开始，后跟标记 “(”、布尔表达式、标记 “)” 和嵌入的语句。

当非结束符有多于一个的扩展时，列出这些可能的扩展（每个扩展单独占一行）。例

如，产生式

```
statement-list: (语句列表:)  
    statement (语句)  
    statement-list statement (语句列表 语句)
```

定义了一个 `statement-list` (语句列表)，它可能是由一个 `statement` (语句) 组成，或者由一个语句列表和随后跟着的一个语句组成。也就是说，该定义是递归的，并且规定了由一个或多个语句组成的语句列表。

写在下方的下标 `opt` 标明了-个可选符号。产生式

```
block:  
    {statement-listopt} ({语句列表可选})
```

是以下产生式

```
block: (块:)  
    {}  
    {statement-list} ({ 语句列表 })
```

的简写形式。它定义了一个块，此块由一个用 “[” 和 “]” 标记括起来的可选语句列表组成。

可选项通常出现在单独的-行中；但当有多个可选项时，那么短语“下列之一 (one of)”将列于在单一-行上的扩展列表之前。这是在一个单独的-行上列出每个可选项的简短形式。例如，下面的产生式

```
real-type-suffix:one of (实数类型后缀: 下列之一)  
    F    f    D    d    M    m
```

是下面的形式

```
real-type-suffix: (实数类型后缀:)  
    F  
    f  
    D  
    d  
    M  
    m
```

的简写。

2.2.2 词法文法

在§2.3、§2.4 和§2.5 中介绍了 C#的词法文法。词法文法的结束符号是 Unicode 字符集中的字符，词法文法规定了字符如何组合以构成标记 (§2.4)、空白 (§2.3.2)、注释 (§2.3.3) 和预处理指令 (§2.5)。

C#程序中的每个源文件必须符合词法文法 (§2.3) 的输入产生式。

2.2.3 句法文法

本章后续的章节和附录将介绍 C#的句法文法。句法文法的结束符号是由词法文法定义

的标记，句法文法指定如何将这此标记组合在一起，以构成完整的 C# 序。

C# 程序中的每个源文件都必须符合句法文法的编译单元产生式（参见 § 9.1）。

2.3 词法分析

输入产生式定义了 C# 源文件的词法结构。C# 程序中的每个源文件必须符合这个词法产生式。

```
input:
    input-sectionopt
input-section:
    input-section-part
    input-section input-section-part
input-section-part:
    input-elementsopt new-line
    pp-directive
input-elements:
    input-element
    input-elements input-element
input-element:
    whitespace
    comment
    token
```

5 个基本元素组成了 C# 源文件的词法结构：行结束符（§ 2.3.1）、空白（§ 2.3.2）、注释（§ 2.3.3）、标记（§ 2.4）和预处理指令（§ 2.5）。在这 5 个基本元素当中，只有标记在 C# 程序的句法中（§ 2.2.3）是至关重要的。

对 C# 源文件的词法处理就是将文件缩减成标记序列，该序列即成为句法分析的输入。行结束符、空白和注释可用于分隔标记，预处理指令可导致跳过源文件中的某些节，除此之外这些词法元素对 C# 程序的句法结构没有任何影响。

在 C# 源文件中，当某些词法产生式匹配一个字符序列时，词法处理总是构成最长的可能的词法元素。例如，字符序列 “//” 被处理为一个单行注释的开始，因为这个词法元素比单个 “/” 标记长一些。

2.3.1 行结束符

行结束符将 C# 源文件分成多行。

new-line:（新行：）

Carriage-return character (U+000D)（回车符）

Line-feed character (U+000A)（换行符）

Carriage-return character (U+000D) followed by **line-feed character** (U+000A)（回车符后跟换行符）

Line-separator character (U+2028)（行分隔符）

Paragraph-separator character (U+2029)（段落分隔符）

某些源代码编辑工具添加了文件尾记号（marker），以使源文件可以被当做合适的结束

行序列来查看。为了保持与这些工具的兼容性，可以在 C# 程序的每个源文件中应用以下转换。

- 如果源文件的最后一个字符是 Control-Z 字符 (U+001A)，就删除它。
- 如果源文件非空，并且如果源文件的最后一个字符不是回车符 (U+000D)、换行符 (U+000A)、行分隔符 (U+2028) 或段落分隔符 (U+2029)，那么在这个源文件的末尾添加回车字符 (U+000D)。

2.3.2 空白

空白使用 Unicode 类 Z 被定义作为任何字符 (包括空白字符)，以及水平制表符、垂直制表符和换页符 (formfeed)。

Whitespace: (空白:)

任何含 Unicode 类 Z 的字符

Horizontal tab character (U+0009) (水平制表符)

Vertical tab character (U+000B) (垂直制表符)

Form-feed character (U+000C) (换页符)

2.3.3 注释

C# 支持两种形式的注释：**单行注释 (single-line comment)** 和 **带分隔符注释 (delimited comment)**。单行注释以字符 “//” 开始，并延续到该源文件行的末尾。带分隔符注释以字符 “/*” 开始，以 “*/” 字符结束。带分隔符注释可以跨多行。

comment:

single-line-comment

delimited-comment

single-line-comment:

// input-characters_{opt}

input-characters:

input-character

input-characters input-character

input-character:

Any Unicode character except a new-line-character (除新行外的任何 Unicode 字符)

new-line-character:

Carriage-return character (U+000D)

Line-feed character (U+000A)

Line-separator character (U+2028)

Paragraph-separator character (U+2029)

delimited-comment:

/* delimited-comment-characters_{opt} */

delimited-comment-characters:

delimited-comment-character

delimited-comment-characters delimited-comment-character

delimited-comment-character:

not-asterisk

*not-slash

not-asterisk:

Any Unicode character except* (除 “*” 外的任何 Unicode 字符)

Not-slash:

Any Unicode character except/ (除 “/” 外的任何 Unicode 字符)

注释并不嵌套。字符序列 “/*” 和 “*/” 在 “//” 注释中没有任何特殊含义，字符序列 “//” 和 “/*” 在带分隔符的注释中没有任何特殊含义。

在字符和字符串内，注释不会被处理。

下面的示例包括一个带分隔符的注释：

```
/*Hello,world program
This program writes "hello ,world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello,world");
    }
}
```

下面的示例展示了几个单行注释。

```
//Hello,world program
//This program writes "hello,world" to the console
//
class Hello //any name will do for this class
{
    static void Main(){ //this method must be named "Main"
        System.Console.WriteLine("hello ,world");
    }
}
```

2.4 标记

标记 (token) 有若干种：标识符、关键字、文字、运算符和标点。空白和注释不是标记，但它们充当标记的分隔符。

token: (标记:)

identifier (标识符)

keyword (关键字)

integer-literal (整数)

real-literal (实数)

character-literal (字符)

string-literal (字符串)

operator-or-punctuator (运算符或标点)

2.4.1 Unicode 字符转义序列

Unicode 字符转义序列表示一个 Unicode 字符。Unicode 字符转义序列在标识符 (§ 2.4.2)、字符 (§ 2.4.4.4) 和规则字符串 (§ 2.4.4.5) 中被处理。不在其他任何位置处理 Unicode 字符转义 (例如在构成运算符、标点符号或关键字时)。

unicode-escape-sequence: (Unicode 转义序列:)

`\u hex-digit hex-digit hex-digit hex-digit`

`\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit`

Unicode 转义序列表示由“\u”或“\U”字符后面的十六进制数字构成的单个 Unicode 字符。由于 C# 在字符和字符串值中使用 Unicode 代码点的 16 位编码,因此从 U+10000 到 U+10FFFF 的 Unicode 字符不能在字符中使用,在字符串中则用一个 Unicode 代理项对来表示,不支持代码数据点在 0x10FFFF 以上的 Unicode 字符。

多次转换不会执行。例如,字符串值“\u005Cu005C”等价于“\u005C”,而不是“\”。Unicode 值“\u005C”表示字符“\”。

下面的示例展示了\u0066的几种用法,它是字符“f”的转义序列。

```
class Class1
{
    static void Test(bool \u0066){
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

以上这个程序等价于下面的程序:

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

2.4.2 标识符

本节给出的标识符规则完全符合 Unicode 标准附件 15 推荐的规则,但以下情况除外:允许将下划线用做初始字符(这是 C 编程语言的传统),允许在标识符中使用 Unicode 转义序列,以及允许“@”字符作为前缀以使关键字能够用做标识符。

identifier: (标识符:)

available-identifier (可用的标识符)

@ identifier-or-keyword (@ 标识符或关键字)

available-identifier: (可用的标识符:)

不是“关键字”的“标识符或关键字”

identifier-or-keyword: (标识符或关键字:)

identifier-start-character identifier-part-characters_{opt} (标识符开始字符 标识符部分字符可选)

identifier-start-character: (标识符开始字符:)

letter-character (字母字符)

_ (下划线字符 U+005F)

identifier-part-characters: (标识符部分字符:)

identifier-part-character (标识符部分字符)

identifier-part-characters identifier-part-character (标识符部分字符 标识符部分字符)

identifier-part-character: (标识符部分字符:)

letter-character (字母字符)

decimal-digit-character (十进制数字字符)

connecting-character (连接字符)

combining-character (组合字符)

formatting-character (格式设置字符)

letter-character: (字母字符:)

类 Lu, Ll, Lt, Lm, Lo 或 Nl 的 Unicode 字符

表示类 Lu, Ll, Lt, Lm, Lo 或 Nl 的字符的 unicode 转义序列

combining-character: (组合字符:)

类 Mn 或 Mc 的 Unicode 字符

表示类 Mn 或 Mc 的字符的 unicode 转义序列

decimal-digit-character: (十进制数字字符:)

类 Nd 的 Unicode 字符

表示类 Nd 的字符的 unicode 转义序列

connecting-character: (连接字符:)

类 Pc 的 Unicode 字符

表示类 Pc 的字符的 unicode 转义序列

formatting-character: (格式设置字符:)

类 Cf 的 Unicode 字符

表示类 Cf 的字符的 unicode 转义序列

关于 Unicode 字符类的信息, 请参考“The Unicode Standard, version 3.0”, 第 4.5 节。

有效标识符示例包括 identifier1, _identifier2 和 @if。

符合规范的程序中的标识符必须符合由“Unicode 标准化格式 C”(按“Unicode 标准附录 15”中的定义)定义的规范格式。当遇到非“标准化格式 C”格式的标识符时,

怎样处理它可由 C 的具体实现确定，但是不要求诊断。

使用前缀 “@” 可以将关键词作为标识符而使用，这在与其他编程语言建立接口时很有用。字符 “@” 实际上不是标识符的一部分，因此在其他语言中可能将此标识符视为不带前缀的正常标识符。带有 “@” 前缀的标识符称为**逐字标识符 (verbatim identifier)**。在非关键字的标识符上使用前缀 “@” 是允许的，但为了风格的一致强烈建议不这样使用。

下面的示例定义了一个名为 class 的类，它有一个名为 static 的静态方法，接受一个命名的 bool 参数。

```
class @class
{
    public static void @static (bool @bool){
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        c1\u0061ss.st\u0061tic(true);
    }
}
```

注意，由于 Unicode 转义字符在关键字中是不允许的，因此标记 c1\u0061ss 是一个标识符，与@class 是同一个标识符。

如果两个标识符在依次应用下面的转换后是一样的，那么它们就被认为是相同的。

- 如果使用了前缀 “@”，删除它。
- 将每个 unicode-escape-sequence (unicode 转义序列) 转换成对应的 Unicode 字符。
- 删除任何 formatting-character (格式字符)。

包含两个连续下划线字符 (U+005F) 的标识符被保留供具体实现使用。例如，一种实现可能是提供以两个下划线开头的可扩展关键字。

2.4.3 关键字

关键字是类似标识符的保留的字符序列，它被保留，并且不能用做标识符，除非加上前缀字符 “@”。

keyword: one of (关键字: 下列之一)

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally

fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

在文法的某些地方，特定的标识符有特别的意义，但不是关键字。例如，在一个属性声明中，`get` 和 `set` 标识符就有特别的意义（§ 10.6.2）。在这些位置，不允许除了 `get` 或 `set` 之外的标识符，因此这种用法不会与将它们用做标识符产生冲突。

2.4.4 文本

文本（**literal**）是值的源代码表述。

- literal: （文本：）
 - boolean-literal （布尔值）
 - integer-literal （整数）
 - real-literal （实数）
 - character-literal （字符）
 - string-literal （字符串）
 - null-literal （空值）

2.4.4.1 布尔值

布尔值有两个：`true` 和 `false`。
boolean-literal: （布尔值：）
`true`
`false`
布尔值的类型是 `bool`。

2.4.4.2 整数

整数用于编写类型 `int`, `uint`, `long` 和 `ulong` 的值。整数有两种可能的形式：十进制和十六进制。
integer-literal: （整数：）

- decimal-integer-literal （十进制整数）
- hexadecimal-integer-literal （十六进制整数）

decimal-integer-literal: (十进制整数:)

decimal-digits integer-type-suffix_{opt} (十进制数字 整数类型后缀可选)

decimal-digits: (十进制数字:)

decimal-digit (十进制数字)

decimal-digits decimal-digit (十进制数字 十进制数字)

decimal-digit: one of (十进制数字: 下列之一)

0 1 2 3 4 5 6 7 8 9

integer-type-suffix: one of (整数类型后缀: 下列之一)

U u L l UL Ul uL ul LU Lu lU lu

hexadecimal-integer-literal: (十六进制整数:)

0x hex-digits integer-type-suffix_{opt} (0x 十六进制数字 整型后缀可选)

0X hex-digits integer-type-suffix_{opt} (0X 十六进制数字 整型后缀可选)

hex-digits: (十六进制数字:)

hex-digit (十六进制数字)

hex-digits hex-digit (十六进制数字 十六进制数字)

hex-digit: one of (十六进制数字: 下列之一)

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

整数的类型是按如下内容确定的。

- 如果文本没有后缀, 则它属于以下所列的类型中第一个能够表示其值的那个类型: int, uint, long 和 ulong。
- 如果文本以 U 或 u 为后缀, 则它属于以下所列的类型中第一个能够表示其值的那个类型: uint, ulong。
- 如果文本以 L 或 l 为后缀, 则它属于以下所列的类型中第一个能够表示其值的那个类型: long, ulong。
- 如果文本以 UL, Ul, uL, ul, LU, Lu, Lu 或 lu 为后缀, 它的类型是 ulong。

如果有一个整型文本表示的值超出了 ulong 类型的范围, 将会出现编译时错误。

为了风格的一致, 在编写 long 型的文本时, 推荐使用 L 代替, 因为字符 l 和数字 1 很容易混淆。

为了允许尽可能小的 int 和 long 值写为十进制整数, 于是就有下面的两个规则。

- 当具有值 2 147 483 648 (2^{31}) 且没有“整型后缀”的一个十进制整数作为标记紧接在一元负运算符标记 (§ 7.6.2) 后出现时, 结果为具有值 -2 147 483 648 (-2^{31}) 的 int 类型常数。在所有其他情况下, 这样的十进制整数属于 uint 类型。
- 当具有值 9 223 372 036 854 775 808 (2^{63}) 的一个十进制整数 (没带整型后缀, 或带有整型后缀 L 或 l) 作为一个标记紧跟在一个一元负运算符标记 (§ 7.6.2) 后出现时, 结果是具有值 -9 223 372 036 854 775 808 (-2^{63}) 的 long 类型的常数。在所有其他情况下, 这样的十进制整数属于 ulong 类型。

2.4.4.3 实数

实数用于编写 float, double 和 decimal 类型的值。

real-literal: (实数:)

decimal-digits . decimal-digits exponent-part_{opt} real-type-suffix_{opt} (十进制数字 . 十进制数字 指数部分可选 实数类型后缀可选)

. decimal-digits exponent-part_{opt} real-type-suffix_{opt} (. 十进制数字 指数部分可选 实数类型后缀可选)

decimal-digits exponent-part real-type-suffix_{opt}(十进制数字 指数部分 实数类型后缀可选)

decimal-digits real-type-suffix (十进制数字 实数类型后缀)

exponent-part: (指数部分:)

e sign_{opt} decimal-digits (e 符号_{可选} 十进制数字)

E sign_{opt} decimal-digits (E 符号_{可选} 十进制数字)

sign: one of (符号: 下列之一)

+ -

real-type-suffix: one of (实数类型后缀: 下列之一)

F f D d M m

如果没有指定 real-type-suffix, 实数文本的类型就是 double。否则, 由实数文本的后缀决定实数文本的类型。

- 以 F 或 f 为后缀的实数文本是 float 类型。例如, 文本 1f, 1.5f, 1e10f 和 123.456F 都是 float 类型。
- 以 D 或 d 为后缀的实数文本是 double 类型。例如, 文本 1d, 1.5d, 1e10d 和 123.456D 都是 double 类型。
- 以 M 或 m 为后缀的实数文本是 decimal 类型。例如, 文本 1m, 1.5m, 1e10m 和 123.456M 都是 decimal 类型。此实数通过取精确值转换为 decimal 值, 如果有必要, 用银行家舍入法 (banker's rounding) (§4.1.7) 舍入为最接近的可表示值, 保留该实数的所有小数位数, 除非值被舍入或者值为零 (在后一种情况中, 符号和小数位数为 0)。因此, 实数 2.900m 经分析后将形成这样的小数: 符号为 0、系数为 2900, 小数位数为 3。

如果给定的文本不能以指定的类型表示, 将会产生编译时错误。

使用 IEEE “就近舍入”模式确定类型 float 或 double 的实数的值。

注意在实数中, 小数点后必须始终是十进制数字。例如, 1.3F 是实数但 1.F 不是。

2.4.4.4 字符

字符表示一个单一字符, 通常由置于引号中的一个字符构成, 例如, 'a'。

character-literal: (字符:)

' character ' (' 字符 ')

character: (字符:)

single-character (单字符)

simple-escape-sequence (简单转义序列)

hexadecimal-escape-sequence (十六进制转义序列)

unicode-escape-sequence (unicode 转义序列)

single-character: (单字符:)

除 ' (U+0027)、\ (U+005C) 和换行符外的任何字符

simple-escape-sequence: one of (简单转义序列: 下列之一)

\' \" \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence: (十六进制转义序列:)

\x hex-digit hex-digit_{opt} hex-digit_{opt} hex-digit_{opt} (\x 十六进制数字 十六进制数字_{可选} 十六进制数字_{可选} 十六进制数字_{可选})

“字符”中紧随反斜线字符 (\) 之后的字符必须是如下字符: ', ", \, 0, a, b, f, n, r, t, u, U, x 或 v, 否则将会出现编译时错误。

十六进制转义序列表示单个 Unicode 字符, 它的值由 “\x” 后接十六进制数组成。如果一个字符所表示的值大于 U+FFFF, 将会产生编译时错误。

在字符中的 Unicode 字符转义序列 (§ 2.4.1), 范围必须在 U+0000 到 U+FFFF 之间。一个简单的转义序列表示一个 Unicode 字符编码, 如下表 2.1 所示。

表 2.1 转义序列与 Unicode 字符编码

Escape Sequence (转义序列)	Character Name (字符名字)	Unicode Encoding (Unicode 编码)
\'	Single quote (单引号)	0x0027
\"	Double quote (双引号)	0x0022
\\	Backslash (反斜线)	0x005C
\0	Null (空)	0x0000
\a	Alert (报警)	0x0007
\b	Backspace (退格)	0x0008
\f	Form feed (换页)	0x000C
\n	New line (换行)	0x000A
\r	Carriage return (回车)	0x000D
\t	Horizontal tab (水平制表符)	0x0009
\v	Vertical tab (垂直制表符)	0x000B

字符的类型为 char。

2.4.4.5 字符串

C#支持两种形式的字符串: 规则字符串 (regular string literal) 和逐字字符串 (verbatim string literal)。

规则字符串由零或多个封闭在双引号中的字符 (例如 "hello") 组成, 并且可以包含简单转义序列 (例如, 制表符序列 "\t") 和十六进制, 以及 Unicode 转义序列。

逐字字符串由一个 "@" 字符紧随其后的双引号字符和零或多个字符组成, 然后是闭合双引号字符。一个简单的例子就是 @ "hello"。在一个逐字字符串中, 分隔符之间的字符是逐字解释的; 惟一的例外就是引号转义序列 (quote-escape-sequence)。特别需要注意的一点是, 简单转义序列、十六进制和 Unicode 转义序列, 在逐字字符串中不会被处理。逐字字符串可以跨多行。

string-literal: (字符串:)

regular-string-literal (规则字符串)
 verbatim-string-literal (逐字的字符串)
 regular-string-literal: (规则字符串:)
 " regular-string-literal-characters_{opt} " (" 规则字符串字符_{可选} ")
 regular-string-literal-characters: (规则字符串字符:)
 regular-string-literal-character (规则字符串字符)
 regular-string-literal-characters regular-string-literal-character (规则字符串字符 规则字符串字符)
 regular-string-literal-character: (规则字符串字符:)
 single-regular-string-literal-character (单个规则字符串字符)
 simple-escape-sequence (简单转义序列)
 hexadecimal-escape-sequence (十六进制转义序列)
 unicode-escape-sequence (unicode 转义序列)
 single-regular-string-literal-character: (单个规则字符串字符:)
 除 " (U+0022)、\ (U+005C) 和换行符外的任何字符
 verbatim-string-literal: (逐字的字符串:)
 @" verbatim-string-literal-characters_{opt} " (@ 逐字的字符串字符_{可选} ")
 verbatim-string-literal-characters: (逐字的字符串字符:)
 verbatim-string-literal-character (逐字的字符串字符)
 verbatim-string-literal-characters verbatim-string-literal-character (逐字的字符串字符 逐字的字符串字符)
 verbatim-string-literal-character: (逐字的字符串字符:)
 single-verbatim-string-literal-character (单个逐字的字符串字符)
 quote-escape-sequence (引号转义序列)
 single-verbatim-string-literal-character: (单个逐字的字符串字符:)
 除 " 外的任何字符
 quote-escape-sequence: (引号转义序列:)
 " "

规则字符串字符中跟在反斜杠字符 (\) 后面的字符必须是下列字符之一: ', ", \, 0, a, b, f, n, r, t, u, U, x 或 v, 否则将发生编译时错误。

下面的例子展示了字符串的一些变体。

```
string a="hello,world";           //hello,world
string b=@"hello,world";         //hello,world
string c="hello \t world";       //hello world
string d=@"hello \t world";     //hello \t world
string e="Joe said \"Hello \" to me"; //Joe said "Hello" to me
string f=@"Joe said \"hello\" to me"; //Joe said "Hello" to me
string g="\\\\server\\share\\file.txt"; //\\server\\share\\file.txt
string h=@"\\server\\share\\file.txt"; //\\server\\share\\file.txt
string i="one\r\ntwo\r\nthree";
string j=@"one
two
three";
```


最后一个字符串 `j` 是一个跨多行的逐字字符串。引号之间的字符（包括空白，如换行符等）也逐字符保留。

由于十六进制转义序列可以有可变数目的十六进制数字，因此字符串 `"\x123"` 只包含一个具有十六进制值 123 的字符。若要创建一个包含具有十六进制值 12 的字符，后跟一个字符 3 的字符串，可以改写为 `"\x00123"` 或 `"\x12" + "3"`。

“字符串”的类型是 `string`。

每个字符串不一定产生新的字符串实例。当根据字符串相等运算符 (§7.9.7) 确认为相等的两个或多个字符串出现在同一个程序集中时，这些字符串引用相同的字符串实例。例如下面程序的输出为 `True`，因为两个字符串引用相同的字符串实例。

```
class Test
{
    static void Main(){
        object a="hello";
        object b="hello";
        System.Console.WriteLine(a==b);
    }
}
```

2.4.4.6 空文本

`null-literal`: (空文本:)
`null`
空文本的类型是 `null` 类型。

2.4.5 运算符和标点

有多种运算符和标点。运算符用于在表达式中描述一个或多个操作数相关的操作。例如，表达式 `"a+b"` 是用 `+` 运算符对两个操作数 `a` 和 `b` 进行相加。标点是用来分组和分隔的。

`operator-or-punctuator`: one of (运算符和标点: 下列之一)

```
{    }    [    ]    (    )    .    '    :    ;
+    -    *    /    %    &    |    ^    !    ~
=    <    >    ?    ++    --    &&    ||    <<    >>
==    !=    <=    >=    +=    -=    *=    /=    %=    &=
|=    ^=    <<=    >>=    ->
```

2.5 预处理指令

预处理指令提供了有条件地忽略源代码某些部分、报告错误和警告条件，以及描绘源代码的不同区域的能力。使用术语“预处理指令”只是为了与 C 和 C++保持一致。在 C# 中没有单独的预处理步骤；预处理指令作为词法分析阶段的一部分来处理。

`pp-directive`: (`pp` 指令:)

pp-declaration (pp 声明)
 pp-conditional (pp 条件)
 pp-line (pp 行)
 pp-diagnostic (pp 诊断)
 pp-region (pp 区域)

下面是可用的预处理指令:

- **#define** 和 **#undef**, 依次用来定义和取消定义条件编译符号的 (§ 2.5.3);
- **#if**, **#elif**, **#else** 和 **#endif**, 用来按条件略过源代码的某些部分 (§ 2.5.4);
- **#line**, 用于控制在发布错误和警告时的行号 (§ 2.5.7);
- **#error** 和 **#warning**, 依次用来引发错误和警告 (§ 2.5.5);
- **#region** 和 **#endregion**, 用于显式标记源代码的某些部分 (§ 2.5.6)。

预处理指令总是单独地占用源代码的一行, 并且总是以“#”字符和指令符名字开始。在“#”字符前面或者“#”字符与指令符之间可以有空白符。

预处理指令不是标记, 也不是 C#的句法文法的一部分。但预处理指令可以用于包含或排除标记序列, 并且以这种方式影响 C#程序的含义。例如, 编译后下面的程序:

```
#define A
#undef B
class C
{
    #if A
        void F() {}
    #else
        void G() {}
    #endif
    #if B
        void H() {}
    #else
        void I() {}
    #endif
}
```

产生与下面的程序完全相同的标记序列:

```
class C
{
    void F() {}
    void I() {}
}
```

因此, 尽管上述两个程序在词法分析中完全不同, 但它们在句法分析中是相同的。

2.5.1 条件编译符号

由**#if**、**#elif**、**#else** 和**#endif** 指令提供的条件编译功能, 是通过预处理表达式 (§ 2.5.2) 和条件编译符号来控制的。

conditional-symbol: (条件符号:)

除 **true** 和 **false** 外的任何标识符或关键字

条件编译符号有两种可能的状态：已定义的或未定义的。在源文件词法处理开始时，条件编译符号除非已由外部机制（如命令行编译器选项）显式定义，否则是未定义的。当处理 `#define` 指令时，在指令中指定的条件编译符号在那个源文件中成为已定义的。此后，该符号就一直保持已定义的状态，直到处理一条关于同一符号的 `#undef` 指令，或者到达源文件的结尾。这意味着一个源文件中的 `#define` 和 `#undef` 指令对同一程序中的其他源文件没有任何影响。

当在预处理表达式中引用时，已定义的条件编译符号具有布尔值 `true`，未定义的条件编译符号具有布尔值 `false`。不要求在预处理表达式中引用条件编译符号之前显式声明它们。相反，未声明的符号只是未定义的，因此具有值 `false`。

条件编译符号的命名空间与 C# 程序中的所有其他命名实体截然不同。只能在 `#define` 和 `#undef` 指令及预处理表达式中引用条件编译符号。

2.5.2 预处理表达式

预处理表达式可以出现在 `#if` 和 `#elif` 指令中。在预处理表达式中允许使用 `!`、`==`、`!=`、`&&` 和 `||` 运算符，并且可以使用括号进行分组。

pp-expression: (pp 表达式:)

whitespace_{opt} pp-or-expression whitespace_{opt} (空白可选 pp 或表达式 空白可选)

pp-or-expression: (pp 或表达式:)

pp-and-expression (pp 与表达式)

pp-or-expression whitespace_{opt} || whitespace_{opt} pp-and-expression (pp 或表达式 空白_{可选} || 空白_{可选} pp 与表达式)

pp-and-expression: (pp 与表达式:)

pp-equality-expression (pp 相等表达式)

pp-and-expression whitespace_{opt} && whitespace_{opt} pp-equality-expression (pp 与表达式 空白_{可选} && 空白_{可选} pp 相等表达式)

pp-equality-expression: (pp 相等表达式:)

pp-unary-expression (pp 一元表达式)

pp-equality-expression whitespace_{opt} == whitespace_{opt} pp-unary-expression (pp 相等表达式 空白_{可选} == 空白_{可选} pp 一元表达式)

pp-equality-expression whitespace_{opt} != whitespace_{opt} pp-unary-expression (pp 相等表达式 空白_{可选} != 空白_{可选} pp 一元表达式)

pp-unary-expression: (pp 一元表达式:)

pp-primary-expression (pp 基本表达式)

! whitespace_{opt} pp-unary-expression (! 空白_{可选} pp 一元表达式)

pp-primary-expression: (pp 基本表达式:)

true

false

conditional-symbol (条件符号)

(whitespace_{opt} pp-expression whitespace_{opt}) ((空白_{可选} pp 表达式 空白_{可选}))

当在预处理表达式中引用时，已定义的条件编译符号具有布尔值 true，未定义的条件编译符号具有布尔值 false。

预处理表达式的计算总是产生一个布尔值。预处理表达式的计算规则与常数表达式 (§7.15) 相同，惟一的例外是：在这里，惟一可引用的用户定义实体是条件编译符号。

2.5.3 声明指令

声明指令用于定义或取消定义条件编译符号。

pp-declaration: (pp 声明:)

whitespace_{opt} # whitespace_{opt} define whitespace conditional-symbol pp-new-line
(空白_{可选} # 空白_{可选} define 空白 条件符号 pp 新行)

whitespace_{opt} # whitespace_{opt} undef whitespace conditional-symbol pp-new-line
(空白_{可选} # 空白_{可选} undef 空白 条件符号 pp 新行)

pp-new-line: (pp 新行:)

whitespace_{opt} single-line-comment_{opt} new-line (空白_{可选} 单行注释_{可选} 新行)

对 **#define** 指令的处理使给定的条件编译符号成为已定义的（从跟在指令后面的源代码行开始）。类似地，对 **#undef** 指令的处理使给定的条件编译符号成为未定义的（从跟在指令后面的源代码行开始）。

源文件中的任何 **#define** 和 **#undef** 指令都必须出现在源文件中第一个“标记” (§2.4) 的前面，否则将发生编译时错误。直观地讲，**#define** 和 **#undef** 指令必须位于源文件中所有“实代码”的前面。

下面的示例是有效的，这是因为 **#define** 指令位于源文件中第一个标记(namespace 关键字) 的前面。

```
#define Enterprise
#if Professional || Enterprise
    #define Advanced
#endif
namespace Megacorp.Data
{
    #if Advanced
        class PivotTable {...}
    #endif
}
```

下面的示例产生编译时错误，因为 **#define** 指令在实代码后面出现。

```
#define A
namespace N
{
    #define B
    #if B
        class Class1 {}
    #endif
}
```

`#define` 指令可用于重复地定义一个已定义的条件编译符号，而不必对该符号插入任何 `#undef`。下面的示例定义一个条件编译符号 `A`，然后再次定义它。

```
#define A
#define A
```

`#undef` 指令可用于取消定义一个本来已经是未定义的条件编译符号。下面的示例定义一个条件编译符号 `A`，然后两次取消定义该符号；第二个 `#undef` 没有作用但仍是有效的。

```
#define A
#undef A
#undef A
```

2.5.4 条件编译指令

条件编译指令用于按条件包含或排除源文件中的某些部分。

`pp-conditional`: (pp 条件:)

```
pp-if-section  pp-elif-sectionsopt  pp-else-sectionopt  pp-endif (pp if 节  pp
elif 节可选  pp else 节可选  pp endif)
```

`pp-if-section`: (pp if 节:)

```
whitespaceopt  #  whitespaceopt  if  whitespace  pp-expression
pp-new-line  conditional-sectionopt (空白可选  #  空白可选  if  空白  pp 表
达式  pp 新行  条件节可选)
```

`pp-elif-sections`: (pp elif 节:)

```
pp-elif-section (pp elif 节)
pp-elif-sections  pp-elif-section (pp elif 节  pp elif 节)
```

`pp-elif-section`: (pp elif 节:)

```
whitespaceopt  #  whitespaceopt  elif  whitespace  pp-expression
pp-new-line  conditional-sectionopt (空白可选  #  空白可选  elif  空白  pp
表达式  pp 新行  条件节可选)
```

`pp-else-section`: (pp-else 节:)

```
whitespaceopt  #  whitespaceopt  else  pp-new-line  conditional-sectionopt
(空白可选  #  空白可选  else  pp 新行  条件节可选)
```

`pp-endif`:

```
whitespaceopt  #  whitespaceopt  endif  pp-new-line (空白可选  #  空白可选
endif  pp 新行)
```

`conditional-section`: (条件节:)

```
input-section (输入节)
skipped-section (跳过节)
```

`skipped-section`: (跳过节:)

```
skipped-section-part (跳过节部分)
skipped-section  skipped-section-part (跳过节  跳过节部分)
```

skipped-section-part: (跳过节部分:)

skipped-characters_{opt} new-line (跳过字符_{可选} 新行)

pp-directive (pp 指令)

skipped-characters: (跳过字符:)

whitespace_{opt} not-number-sign input-characters_{opt} (空白_{可选} 非数字符号 输入字符_{可选})

not-number-sign: (非数字符号:)

除 # 外的任何输入字符

按照语法的规定, 条件编译指令必须写成指令集的形式。指令集的组成依次为: 一个 #if 指令、一个或多个 #elif 指令(或没有)、一个或多个 #else 指令(或没有)和一个 #endif 指令。

指令之间是源代码的条件节。每节代码直接被位于它前面的那个指令控制。条件节本身可以包含嵌套的条件编译指令, 前提是这些指令构成完整的指令集。

“pp 条件”最多只能选择一个它所包含的“条件节”去做通常的词法处理:

- 按顺序计算 #if 和 #elif 指令的“pp 表达式”直到获得值 true。如果表达式的结果为 true, 则选择对应指令的“条件节”;
- 如果所有“pp 表达式”的结果都为 false 并且存在 #else 指令, 则选择 #else 指令的“条件节”;
- 否则不选择任何“条件节”。

选定的“条件节”(若有)按正常的“输入节”处理: 节中包含的源代码必须符合词法文法; 从节中的源代码生成标记; 节中的预处理指令具有规定的效果。

剩余的“条件节”(若有)按“跳过节”处理: 除了预处理指令, 节中的源代码不必一定要符合词法文法; 不从节中的源代码生成任何词法标记; 节中的预处理指令必须在词法上正确, 但不另外处理。在按“跳过节”处理的“条件节”中, 任何嵌套的“条件节”(包含在嵌套的 #if...#endif 和 #region...#endregion 构造中)也按“跳过节”处理。

下面的示例解释如何嵌套条件编译指令:

```
#define Debug    // Debugging on
#undef Trace     // Tracing off
class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
            #if Trace
                WriteToLog(this.ToString());
            #endif
        #endif
        CommitHelper();
    }
}
```

除预处理指令外, 跳过的源代码与词法分析无关。例如, 尽管在 #else 节中有未结束的注释, 但下面的示例仍然有效:

```

#define Debug    // Debugging on
class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* 做其他事情 */
        #endif
    }
}

```

但请注意，即使是在源代码的跳过节中，也要求预处理指令在词法上正确。
当预处理指令出现在多行输入元素的内部时，不作为预处理指令处理。例如，程序：

```

class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
    world
#else
    Nebraska
#endif
    ");
    }
}

```

输出结果为：

```

hello,
#if Debug
    world
#else
    Nebraska
#endif

```

在特殊情况下，如何处理预处理指令集可能取决于 pp 表达式的计算。下面的示例总是生成同样的标记流（class Q{}），不管是否定义了 X。

```

#if X
    /*
#else
    /* */ class Q { }
#endif

```

如果定义了 X，由于多行注释的缘故，则只处理 #if 和 #endif 指令。如果未定义 X，则这三个指令（#if，#else，#endif）是指令集的组成部分。

2.5.5 诊断指令

诊断指令用于显式生成错误信息和警告消息，这些信息的报告方式与其他编译时错误和警告相同。

pp-diagnostic: (pp 诊断:)

whitespace_{opt} # whitespace_{opt} error pp-message (空白_{可选} # 空白_{可选})


```

error    pp 消息)
whitespaceopt    #    whitespaceopt    warning    pp-message (空白可选    #    空
白可选    warning    pp 消息)
pp-message: (pp 消息:)
new-line (新行)

```

```

whitespace    input-charactersopt    new-line (空白    输入字符可选    新行)

```

下面的示例总是产生一个警告(“Code review needed before check-in”),如果同时定义条件符号 Debug 和 Retail,则产生一个编译时错误(“A build can't be both debug and retail”).

```

#warning Code review needed before check-in
#if Debug && Retail
    #error A build can't be both debug and retail
#endif
class Test {...}

```

注意, pp-message (pp 消息) 可以包含任意文本。特别指出,它可以包含格式不正确的标记,比如“can't”中的单引号就是这样。

2.5.6 区域指令

区域指令用于显式标记源代码的区域。

pp-region: (pp 区域:)

```

pp-start-region    conditional-sectionopt    pp-end-region (pp 开始区域    条件节可选
pp 结束区域)

```

pp-start-region: (pp 开始区域:)

```

whitespaceopt    #    whitespaceopt    region    pp-message (空白可选    #    空白可选
region    pp 消息)

```

pp-end-region: (pp 结束区域:)

```

whitespaceopt    #    whitespaceopt    endregion    pp-message (空白可选    #
空白可选    endregion    pp 消息)

```

区域不具有任何附加的语义含义;区域旨在由程序员或自动工具用来标记源代码中的节。#region 或 #endregion 指令中指定的消息同样不具有任何语义含义,它只是用于标识区域。匹配的 #region 和 #endregion 指令可能具有不同的“pp 消息”。

区域的词法处理:

```

#region
...
#endregion

```

与以下形式的条件编译指令的词法处理完全对应:

```

#if true
...
#endif

```

2.5.7 行指令

行指令可用于改变编译器在输出（如警告和错误）中报告的行号和源文件名称。
行指令最通用于从某些其他文本输入生成 C# 源代码的元编程工具。

```
pp-line: (pp 行: ) ,
    whitespaceopt # whitespaceopt line whitespace line-indicator
pp-new-line (空白可选 # 空白可选 line 空白 行指示符 pp 新行)
line-indicator: (行指示符:)
    decimal-digits whitespace file-name (十进制数字 空白 文件名)
    decimal-digits (十进制数字)
    default
    hidden
```

```
file-name: (文件名:)
    " file-name-characters " (" 文件名字符 ")
file-name-characters: (文件名字符:)
    file-name-character (文件名字符)
    file-name-characters file-name-character (文件名字符 文件名字符)
file-name-character: (文件名字符:)
    除 " 外的任何输入字符
```

当不存在 #line 指令时，编译器在它的输出中报告真实的行号和源文件名称。#line 指令在从某些其他文本输入生成 C# 源代码的元编程工具中最常用。

当处理的 #line 指令包含不是 default 的行指示符时，编译器将该指令“后面”的行视为具有给定的行号（如果指定了，还包括文件名）。

#line default 指令消除前面所有 #line 指令的影响。编译器报告后续行的真实行信息，就像尚未处理任何 #line 指令一样。

#line hidden 指令对错误信息中报告的文件号和行号无效，但对源代码级调试确实有效。调试时，#line hidden 指令和后面的 #line 指令（不是 #line hidden）之间的所有行都没有行号信息。在调试器中逐句执行代码时，将全部跳过这些行。

注意，file-name（文件名）与常规字符串的不同之处在于不处理转义字符：“\”字符在 file-name（文件名）中只表示一个普通的反斜杆字符。

第3章 基本概念

3.1 应用程序启动

具有入口点的程序集称为应用程序。应用程序运行时，将创建新的应用程序域。同一台计算机上可能会同时运行着同一个应用程序的若干个实例，此时，每一个实例都拥有各自的应用程序域。

应用程序域通过作为应用程序状态的容器使应用程序得以隔离。应用程序域作为应用程序中和它使用的类库中所定义的地类型的容器和边界。同一个类型若被加载到不同的应用程序域中就成为各自独立的客体，由它们在各自应用程序域中产生的实例亦不可直接共享。例如，对于这些类型的静态变量，每个应用程序域都有自己的副本，并且这些类型的静态构造函数在每个应用程序域中最多运行一次。关于如何处理程序域的创建和销毁，各实现方法可以按具体情况确定自己的策略或机制。

当执行环境调用指定的方法（称为程序的入口点）时发生**应用程序启动（application startup）**。此入口点方法总是被命名为 **Main**，可以具有下列签名之一：

```
static void Main() {...}
static void Main(string[] args) {...}
static int Main() {...}
static int Main(string[] args) {...}
```

如上所示，入口点可以选择返回一个 **int** 值。此返回值用于应用程序终止（§ 3.2）。

入口点可以包含一个参数（可选）。该参数可以具有任意名称，但参数的类型必须为 **string[]**。如果存在参数，执行环境会创建并传递一个包含命令行参数的 **string[]** 参数，这些命令行参数是在启动应用程序时指定的。**string[]** 参数永远不能为 **null**，但如果没有指定命令行参数，它的长度可以为零。

由于 C# 支持方法重载，因此类或结构可以包含某个方法的多个定义（前提是每个定义有不同的签名）。但在一个程序内，没有任何类或结构可以包含一个以上的名为 **Main** 的方法，因为 **Main** 的定义限定它只能用做应用程序的入口点。允许使用 **Main** 的其他重载版本，前提是它们具有一个以上的参数，或者它们的惟一参数的类型不是 **string[]**。

应用程序可由多个类或结构组成。在这些类或结构中，可能会有若干个拥有自己的 **Main** 方法，因为 **Main** 的定义限定它只能用做应用程序的入口点。这种情况下，必须利用某种外部机制（如命令行编译器的选项）来指定将哪一个 **Main** 方法用做入口点。

在 C# 中，每个方法都必须定义为类或结构的成员。通常，方法的已声明可访问性（§ 3.5.1）由其声明中指定的访问修饰符（§ 10.2.3）确定。同样，类型的已声明可访问性由其声明中指定的访问修饰符确定。为了使给定类型的给定方法可以被调用，类型和成

员都必须是可访问的。然而，应用程序入口点是一种特殊情况。具体而言，执行环境可以访问应用程序的入口点，无论它本身的可访问性和封闭它的类型的可访问性是如何在声明语句中设置的。

在所有其他方面，入口点方法的行为与非入口点方法的行为类似。

3.2 应用程序终止

应用程序终止 (application termination) 将控制返回给执行环境。

如果应用程序的入口点方法的返回类型为 `int`，则返回的值用做应用程序的终止状态代码。此代码的用途是允许与执行环境进行关于应用程序运行状态（成功或失败）的通信。

如果入口点方法的返回类型为 `void`，那么，到达了终止该方法的右大括号 `()`，或者执行不带表达式的 `return` 语句，将产生终止状态代码 0。

在应用程序终止之前，将调用其中还没有被垃圾回收的所有对象的析构函数，除非这类清理功能已被设置为停止使用（例如，通过调用库方法 `GC.SuppressFinalize`）。

3.3 声明

C# 程序中的声明定义程序的构成元素。C# 程序是用命名空间 (§ 9) 组织起来的，一个命名空间可以包含类型声明和嵌套的命名空间声明。类型声明 (§ 9.5) 用于定义类 (§ 10)、结构 (§ 11)、接口 (§ 13)、枚举 (§ 14) 和委托 (§ 15)。在一个类型声明中可以使用哪些类型作为其成员，取决于该类型声明的形式。例如，类声明可以包含常数声明 (§ 10.3)、字段声明 (§ 10.4)、方法声明 (§ 10.5)、属性声明 (§ 10.6)、事件声明 (§ 10.7)、索引器声明 (§ 10.8)、运算符声明 (§ 10.9)、实例构造函数声明 (§ 10.10)、静态构造函数声明 (§ 10.11)、析构函数声明 (§ 10.12) 和嵌套类型声明 (§ 10.2.6)。

声明所定义的名称属于它自己所属的那个**声明空间 (declaration space)**。除非是重载成员 (§ 3.6)，否则，在同一个声明空间下若有两个以上的声明语句声明了具有相同名称的成员，就会产生编译时错误。同一个声明空间内绝不能包含不同类型的同名成员。例如，声明空间绝不能包含同名的字段和方法。

有若干种不同类型的声明空间，如下所述。

- 在程序的所有源文件中，命名空间成员声明 (`namespace-member-declaration`) 若没有被置于任何一个命名空间声明下，它就属于一个组合声明空间[称为全局声明空间 (`global declaration space`)]。
- 在程序的所有源文件中，一个命名空间成员声明若在命名空间声明中具有相同的完全限定的命名空间名称，它就属于一个组合声明空间。
- 每个类、结构或接口声明创建一个新的声明空间。新的声明空间名称是通过类成员声明、结构成员声明或接口成员声明引入的。除了重载实例构造函数声明和静态构造函数声明外，类或结构成员声明不能引入与该类或结构同名的成员。类、结构或接口允许声明重载方法和索引器。另外，类或结构允许重载实例构造函数

和运算符的声明。例如，类、结构或接口可以包含多个同名的方法声明，前提是这些方法声明的签名（§ 3.6）不同。注意，基类与类的声明空间无关，基接口与接口的声明空间无关。因此，允许在派生类或接口内声明与所继承的成员同名的成员。我们说这类成员隐藏（hide）了那些被它们所继承的成员。

- 每个枚举声明创建一个新的声明空间。名称通过枚举成员声明（enum-member-declaration）引入此声明空间。
- 每个块或 switch 块为局部变量和常量创建一个不同的声明空间。名称通过局部变量声明和局部常量声明引入此声明空间。如果块是实例构造函数、方法或运算符声明的体，或是索引器声明的 get 或 set 访问器，那么这些声明中声明的参数是块的局部变量声明空间的成员。块的局部变量声明空间包含任何嵌套块。因此，在嵌套块中不可能声明与封闭它的块中的局部变量同名的局部变量。
- 每个块或 switch 块都为标签创建一个单独的声明空间。名称通过标记语句（labeled-statement）引入此声明空间，通过 goto 语句被引用。块的标签声明空间（label declaration space）包含任何嵌套块。因此，在嵌套块中不可能声明与封闭它的块中的标签同名的标签。

声明名称的文本顺序通常不重要。具体说来，声明和使用命名空间、常数、方法、属性、事件、索引器、运算符、实例构造函数、析构函数、静态构造函数和类型时，文本顺序并不重要。而在下列情况下声明顺序非常重要：

- 字段声明和局部变量声明的声明顺序确定其初始值设定项（若有的话）的执行顺序；
- 在使用局部变量前必须先定义它们（§ 3.7）；
- 当省略常数表达式的值时，枚举成员声明（§ 14.3）的声明顺序非常重要。

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}
namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

以上两个命名空间声明属于相同的声明空间，在本示例中声明两个具有完全限定名 **Megacorp.Data.Customer** 和 **Megacorp.Data.Order** 的类。由于两个声明共同构成同一个声明空间，因此如果每个声明中都包含一个同名类的声明，则将导致编译时错误。

正如前面的详细说明，块的声明空间包括所有嵌套块。因此，在下面的示例中，F 和 G 方法导致编译时错误，因为名称 i 是在外部块中声明的，不能在内部块中重新声明。但方法 H 和 I 都是有效的，因为这两个 i 是在单独的非嵌套块中声明的。

```

class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }
    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }
    void H() {
        if (true) {
            int i = 0;
        }
        if (true) {
            int i = 1;
        }
    }
    void I() {
        for (int i = 0; i < 10; i++)
            H();
        for (int i = 0; i < 10; i++)
            H();
    }
}

```

3.4 成员

命名空间和类型具有**成员 (member)**。通常可以通过使用限定名来访问一个实体的成员；限定名以对该实体的引用开头，后跟一个“.”标记，再接成员的名称。

类型的成员或者是在该类型中声明的，或者是从该类型的基类继承的。当类型从基类继承时，基类的所有成员（实例构造函数、析构函数和静态构造函数除外）都成为派生类型的成员。基类中关于成员的可访问性的声明不能控制该成员是否可继承：继承性适用于任何成员，只要它们不是实例构造函数、静态构造函数或析构函数。然而，在派生类中可能不能访问已被继承的成员，原因或者是因为其已声明可访问性 (§ 3.5.1)，或者是因为它已被类型本身中的声明所隐藏 (§ 3.7.2)。

3.4.1 命名空间成员

命名空间和类型若没有封闭它的命名空间，则属于全局命名空间的成员。这直接对应于全局声明空间中声明的名称。

在某命名空间中声明的命名空间和类型是该命名空间的成员。这直接对应于该命名空间的声明空间中声明的名称。

命名空间没有访问限制。不可能把命名空间设置成私有的、受保护的或内部的，命名空间名称始终是可公开访问的。

3.4.2 结构成员

结构的成员是在结构中声明的成员和从类 `object` 继承的成员。

简单类型的成员直接对应于结构类型的成员，此简单类型正是该结构的化名。

- `sbyte` 的成员是 `System.SByte` 结构的成员。
- `byte` 的成员是 `System.Byte` 结构的成员。
- `short` 的成员是 `System.Int16` 结构的成员。
- `ushort` 的成员是 `System.UInt16` 结构的成员。
- `int` 的成员是 `System.Int32` 结构的成员。
- `uint` 的成员是 `System.UInt32` 结构的成员。
- `long` 的成员是 `System.Int64` 结构的成员。
- `ulong` 的成员是 `System.UInt64` 结构的成员。
- `char` 的成员是 `System.Char` 结构的成员。
- `float` 的成员是 `System.Single` 结构的成员。
- `double` 的成员是 `System.Double` 结构的成员。
- `decimal` 的成员是 `System.Decimal` 结构的成员。
- `bool` 的成员是 `System.Boolean` 结构的成员。

3.4.3 枚举成员

枚举成员是在枚举中声明的常数和从类 `System.Enum` 继承的成员。

3.4.4 类成员

类成员是在类中声明的成员和从该类的基类（没有基类的 `object` 类除外）继承的成员。从基类继承的成员包括基类的常数、字段、方法、属性、事件、索引器、运算符和类型，但不包括基类的实例构造函数、析构函数和静态构造函数。基类成员是否被继承与它们的可访问性无关。

类声明可以包含以下对象的声明：常数、字段、方法、属性、事件、索引器、运算符、实例构造函数、析构函数、静态构造函数和类型。

`object` 和 `string` 的成员直接对应于它们所化名的类类型的成员：

- `object` 的成员是 `System.Object` 类的成员；
- `string` 的成员是 `System.String` 类的成员。

3.4.5 接口成员

接口成员是在接口中和该接口的所有基接口中声明的成员，以及从 `object` 类继承的成员。

3.4.6 数组成员

数组成员是从类 `System.Array` 继承的成员。

3.4.7 委托成员

委托成员是从类 `System.Delegate` 继承的成员。

3.5 成员访问

成员的声明可用于控制对该成员的访问。成员的可访问性是由该成员的可访问性声明 (§ 3.5.1) 和直接包含它的那个类型的可访问性 (若它存在) 结合起来确定的。

如果允许对特定成员进行访问, 则称该成员是“可访问的 (accessible)”。相反, 如果不允许对特定成员进行访问, 则称该成员是“不可访问的 (inaccessible)”。当导致访问发生的源代码的文本位置包括在某成员的可访问域 (§ 3.5.2) 中时, 允许对该成员进行访问。

3.5.1 已声明可访问性

成员的“已声明可访问性”可以是下列之一:

- **public**, 选择它的方法是在成员声明中包括 **public** 修饰符。**public** 的直观含义是“访问不受限制”。
- **protected**, 选择它的方法是在成员声明中包括 **protected** 修饰符。**protected** 的直观含义是“访问范围限定于它所属的类或从该类派生的类型”。
- **internal**, 选择它的方法是在成员声明中包括 **internal** 修饰符。**internal** 的直观含义是“访问范围限定于此程序”。
- **protected internal** (意为受保护或内部的), 选择它的方法是在成员声明中包括 **protected** 和 **internal** 修饰符。**protected internal** 的直观含义是“访问范围限定于此程序或那些由它所属的类派生的类型”。
- **private**, 选择它的方法是在成员声明中包括 **private** 修饰符。**private** 的直观含义是“访问范围限定于它所属的类型”。

只有已声明可访问性的某种类型是允许的, 这依赖于该成员声明出现处的上、下文。此外, 当成员声明不包含任何访问修饰符时, 声明发生处的上、下文会为该成员选择一个默认的已声明可访问性。

- 命名空间隐式地具有 **public** 已声明可访问性。在命名空间声明中不允许使用访问修饰符。
- 编译单元或命名空间中声明的类型可以具有 **public** 或 **internal** 已声明可访问性, 默认的已声明可访问性为 **internal**。
- 类成员可具有 5 种已声明可访问性中的任何一种, 默认为 **private** 已声明可访问

性（请注意，声明为类成员的类型可具有 5 种已声明可访问性中的任何一种，而声明为命名空间成员的类型只能具有 `public` 或 `internal` 已声明可访问性）。

- 结构成员可以具有 `public`, `internal` 或 `private` 已声明可访问性并默认为 `private` 已声明可访问性，这是因为结构是隐式地密封的。结构的成员若是在此结构中声明的（也就是说，不是由该结构从它的基类中继承的），则不能具有 `protected` 或 `protected internal` 已声明可访问性（请注意，声明为结构成员的类型可具有 `public`, `internal` 或 `private` 已声明可访问性，而声明为命名空间成员的类型只能具有 `public` 或 `internal` 已声明可访问性）。
- 接口成员隐式地具有 `public` 已声明可访问性。在接口成员声明中不允许使用访问修饰符。
- 枚举成员隐式地具有 `public` 已声明可访问性。在枚举成员声明中不允许使用访问修饰符。

3.5.2 可访问域

成员的可访问域（**accessibility domain**）由（可能是不连续的）程序文本节组成，从那里可以访问该成员。出于定义成员可访问域的目的，如果成员不是在某个类型内声明的，就称该成员是顶级的（`top level`）；如果成员是在其他类型内声明的，就称该成员是嵌套的。此外，程序的程序文本定义为包含在该程序的所有源文件中的全部程序文本，而类型的程序文本定义为包含在该类型（可能还包括嵌套在该类型内的类型）的“类体”、“结构体”、“接口体”或“枚举体”中的开始和结束（“{”和“}”）标记之间的全部程序文本。

预定义类型（如 `object`, `int` 或 `double`）的可访问域是无限制的。

在程序 `P` 中声明的顶级类型 `T` 的可访问域定义如下：

- 如果 `T` 的已声明可访问性为 `public`，则 `T` 的可访问域是 `P` 的及引用 `P` 的任何程序的程序文本。
- 如果 `T` 的已声明可访问性为 `internal`，则 `T` 的可访问域是 `P` 的程序文本。

从这些定义可以推断出：顶级类型的可访问域始终至少是声明了该类型的程序的程序文本。

在程序 `P` 内的类型 `T` 中声明的嵌套成员 `M` 的可访问域定义如下（注意 `M` 本身可能就是一个类型）：

- 如果 `M` 的已声明可访问性为 `public`，则 `M` 的可访问域是 `T` 的可访问域。
- 如果 `M` 的已声明可访问性是 `protected internal`，则设 `D` 表示 `P` 的程序文本和从 `T` 派生的任何类型（在 `P` 的外部声明）的程序文本的并集。`M` 的可访问域是 `T` 与 `D` 的可访问域的交集。
- 如果 `M` 的已声明可访问性是 `protected`，则设 `D` 表示 `T` 的程序文本和从 `T` 派生的任何类型的程序文本的并集。`M` 的可访问域是 `T` 与 `D` 的可访问域的交集。
- 如果 `M` 的已声明可访问性为 `internal`，则 `M` 的可访问域是 `T` 的可访问域与 `P` 的程序文本的交集。

- 如果 M 的已声明可访问性为 `private`，则 M 的可访问域是 T 的程序文本。

从这些定义可以看出，嵌套成员的可访问域总是至少为声明该成员的类型程序文本，成员的可访问域包含的范围绝不会比声明该成员的类型可访问域更广。

用直观的话来讲，当访问类型或成员 M 时，按下列步骤进行计算以确保允许进行访问：

- 首先，如果 M 是在某个类型内（相对于编译单元或命名空间）声明的，则当该类型不可访问时将会发生编译时错误；
- 然后，如果 M 为 `public`，则允许进行访问；
- 如果 M 为 `protected internal`，则当访问发生在声明了 M 的程序中，或发生在从声明 M 的类派生的类中并通过派生类类型（§ 3.5.3）进行访问时，允许进行访问；
- 如果 M 为 `protected`，则当访问发生在声明了 M 的类中，或发生在从声明 M 的类派生的类中并通过派生类类型（§ 3.5.3）进行访问时，允许进行访问；
- 如果 M 为 `internal`，则当访问发生在声明了 M 的程序中时允许进行访问；
- 如果 M 为 `private`，则当访问发生在声明了 M 的类型中时允许进行访问；
- 否则，类型或成员不可访问，并发生编译时错误。

下面是一个示例：

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}
internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;
    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

在该示例中，类和成员具有下列可访问域：

- A 和 A.X 的可访问域无限制；
- A.Y, B, B.X, B.Y, B.C, B.C.X 和 B.C.Y 的可访问域是包含程序的程序文本；
- A.Z 的可访问域是 A 的程序文本；
- B.Z 和 B.D 的可访问域是 B 的程序文本，包括 B.C 和 B.D 的程序文本；
- B.C.Z 的可访问域是 B.C 的程序文本；

- B.D.X 和 B.D.Y 的可访问域是 B 的程序文本，包括 B.C 和 B.D 的程序文本；
- B.D.Z 的可访问域是 B.D 的程序文本。

从示例可看出，成员的可访问域绝不会大于包含它的类型的可访问域。例如，即便是所有的 X 成员都具有公共级的已声明可访问性，除了 A.X 外，所有其他成员的可访问域都受包含类型的约束。

如 § 3.4 中所描述的那样，基类的所有成员（实例构造函数、析构函数和静态构造函数除外）都由派生类型继承，其中甚至包括基类的私有成员。但是，私有成员的可访问域只包括声明该成员的类型程序文本。看看下面的示例：

```
class A
{
    int x;
    static void F(B b) {
        b.x = 1;    // Ok
    }
}
class B: A
{
    static void F(B b) {
        b.x = 1;    // 错误, x 不可访问
    }
}
```

其中，类 B 继承类 A 的私有成员 x。因为该成员是私有的，所以只能在 A 的“类体”中对它进行访问。因此，对 b.x 的访问在 A.F 方法中取得了成功，在 B.F 方法中却失败了。

3.5.3 实例成员的受保护访问

当在声明了某个 **protected** 实例成员的类的程序文本之外访问该实例成员时，以及当在包含有某个 **protected internal** 实例成员声明的程序的程序文本之外访问该实例成员时，要求这种访问发生在该成员所属类的一个派生类的程序文本中，并引用该派生类的一个实例。也就是说，假定 B 是一个基类，它声明了一个受保护的实例成员 M，并且假定 D 是从 B 派生的类。在 D 的“类体”中，对 M 的访问可采取下列形式之一：

- M 形式的非限定类型名或初等表达式；
- E.M 形式的初等表达式（其中，E 是类 D 或是从 D 派生的类）；
- base.M 形式的初等表达式。

除了上述形式外，派生类还可以在它自己的构造函数初始值设定项（§ 10.10.1）访问它的基类的受保护的实例构造函数。

在下面的示例中，在 A 中可以通过 A 和 B 的实例访问 x，这是因为在两种情况下的访问都通过 A 的实例或从 A 派生的类发生。但是在 B 中，由于 A 不从 B 派生，所以不可能通过 A 的实例访问 x。

```
public class A
{
    protected int x;
```

```

        static void F(A a, B b) {
            a.x = 1;    // Ok
            b.x = 1;    // Ok
        }
    }
    public class B: A
    {
        static void F(A a, B b) {
            a.x = 1;    //错误, 必须通过 B 的实例访问
            b.x = 1;    // Ok
        }
    }
}

```

3.5.4 可访问性约束

C# 语言中的有些构造要求某个类型至少与某个成员或其他类型具有同样的可访问性。如果 T 的可访问域是 M 可访问域的超集, 我们就说类型 T 至少与成员或类型 M 具有同样的可访问性。换言之, 如果 T 在可访问 M 的所有上、下文中都是可访问的, 则 T 至少与 M 具有同样的可访问性。

存在下列可访问性约束:

- 类类型的直接基类必须至少与类类型本身具有同样的可访问性。
- 接口类型的显式基接口必须至少与接口类型本身具有同样的可访问性。
- 委托类型的返回类型和参数类型必须至少与委托类型本身具有同样的可访问性。
- 常数的类型必须至少与常数本身具有同样的可访问性。
- 字段的类型必须至少与字段本身具有同样的可访问性。
- 方法的返回类型和参数类型必须至少与方法本身具有同样的可访问性。
- 属性的类型必须至少与属性本身具有同样的可访问性。
- 事件的类型必须至少与事件本身具有同样的可访问性。
- 索引器的类型和参数类型必须至少与索引器本身具有同样的可访问性。
- 运算符的返回类型和参数类型必须至少与运算符本身具有同样的可访问性。
- 实例构造函数的参数类型必须至少与实例构造函数本身具有同样的可访问性。

在下面的示例中, B 类导致编译时错误, 因为 A 并不具有至少与 B 相同的可访问性。

```

class A {...}
public class B: A {...}

```

同样, 在下面的示例中 B 中的方法 H 导致编译时错误, 因为返回类型 A 并不具有至少与该方法相同的可访问性。

```

class A {...}
public class B
{
    A F() {...}
    internal A G() {...}
    public A H() {...}
}

```

3.6 签名和重载

方法、实例构造函数、索引器和运算符是由它们的签名来刻画的：

- 方法签名由方法的名称和它的每一个参数（按从左到右的顺序）的类型和种类（值、引用或输出）组成。需注意的是，方法签名既不包含返回类型，也不包含 `params` 修饰符（它可用于最右边的参数）。
- 实例构造函数签名由它的每一个参数（按从左到右的顺序）的类型和种类（值、引用或输出）组成。具体说来，实例构造函数的签名不包含可为最右边的参数指定的 `params` 修饰符。
- 索引器签名由它的每一个参数（按从左到右的顺序）的类型组成。具体说来，索引器的签名不包含元素类型。
- 运算符签名由运算符的名称和它的每一个参数（按从左到右的顺序）的类型组成。具体说来，运算符的签名不包含结果类型。

签名是对类、结构和接口的成员实施重载的机制：

- 方法重载允许类、结构或接口用同一个名称声明多个方法，条件是它们的签名在该类、结构或接口中是惟一的。
- 实例构造函数重载允许类或结构声明多个实例构造函数，条件是它们的签名在该类或结构中是惟一的。
- 索引器重载允许类、结构或接口声明多个索引器，条件是它们的签名在该类、结构或接口中是惟一的。
- 运算符重载允许类或结构用同一个名称声明多个运算符，条件是它们的签名在该类或结构中是惟一的。

```
interface ITest
{
    void F();                // F()
    void F(int x);           // F(int)
    void F(ref int x);       // F(ref int)
    void F(int x, int y);    // F(int, int)
    int F(string s);         // F(string)
    int F(int x);            // F(int)      错误
    void F(string[] a);      // F(string[])
    void F(params string[] a); // F(string[]) 错误
}
```

请注意，所有 `ref` 和 `out` 参数修饰符（§ 10.5.1）都是签名的组成部分。因此，`F(int)` 和 `F(ref int)` 这两个签名都具有惟一性。还请注意，返回类型和 `params` 修饰符不是签名的组成部分，所以不可能仅基于返回类型或是否存在 `params` 修饰符来实施重载。因此，上面列出的关于方法 `F(int)` 和 `F(params string[])` 的声明会导致编译时错误。

3.7 范围

名称的范围是一个程序文本区域，在其中可以引用由该名称声明的实体，对该名称加

以限定。范围可以嵌套，并且内部范围可以重新声明外部范围中的名称的含义（但这并不会取消 § 3.3 强加的限制，即在嵌套块中不可能声明与它的封闭块中的局部变量同名的局部变量）。

因此，我们说外部范围中的这个同名的名称在由内部范围覆盖的程序文本区域中是隐藏的，只能通过它的限定名才能从内部范围来访问它。

- 由“命名空间成员声明”（§ 9.4）所声明的命名空间成员的范围，如果没有其他封闭它的“命名空间成员声明”，则它的范围是整个程序文本。
- 命名空间声明中命名空间成员声明所声明的命名空间成员的范围是这样定义的，如果该命名空间成员声明的完全限定名为 *N*，则其声明的命名空间成员的范围是：完全限定名为 *N* 或以 *N* 开头的每个命名空间声明的命名空间体。
- 由 `using` 指令（§ 9.3）定义或导入的名称的范围是，出现 `using` 指令的编译单元或命名空间体内的整个“命名空间成员声明”中。`using` 指令可以使零个或多个命名空间或类型名称在特定的编译单元或命名空间体中可用，但不会把任何新成员提供给隐含的下层声明空间。换言之，`using` 指令仅在使用它的编译单元或命名空间体范围内有效，它的功效是不可传递的。
- 由“类成员声明”（§ 10.2）所声明的成员范围是该声明所在的那个“类体”。此外，类成员的范围扩展到该成员的可访问域（§ 3.5.2）中的那些派生类的“类体”。
- 由“结构成员声明”（§ 11.2）声明的成员范围是该声明所在的结构体。
- 由“枚举成员声明”（§ 14.3）声明的成员范围是该声明所在的枚举体。
- 在“方法声明”（§ 10.5）中声明的参数范围是该方法所在的方法体。
- 在“索引器声明”（§ 10.8）中声明的参数范围是该索引器声明的访问器声明。
- 在“运算符声明”（§ 10.9）中声明的参数范围是该运算符声明所在的块。
- 在“构造函数声明”（§ 10.10）中声明的参数范围是该构造函数声明的构造函数初始值设定项和块。
- 在“标记语句”（§ 8.4）中声明的标签范围是该声明所在的块。
- 在“局部变量声明”（§ 8.5.1）中声明的局部变量范围是该声明所在的块。
- 在 `switch` 语句（§ 8.7.2）的 `switch` 块中声明的局部变量范围是 `switch` 块。
- 在 `for` 语句的 `for` 初始值设定项（§ 8.8.3）中声明的局部变量的范围是，该 `for` 语句的 `for` 初始值设定项、`for` 条件、`for` 迭代程序及所包含的语句。
- 在局部常数声明（§ 8.5.2）中声明的局部常量范围是该声明发生的块。在某局部常量常数声明符之前的文本位置中引用该局部常量是编译时错误。

在命名空间、类、结构或枚举成员的范围内，可以在位于该成员的声明之前的文本位置引用该成员。例如：

```
class A
{
    void F() {
        i = 1;
    }
    int i = 0;
}
```

这里，`F` 在声明 `i` 之前引用它是有效的。

在局部变量的范围内，在位于该局部变量的“局部变量声明符”之前的文本位置引用该局部变量是编译时错误。例如

```
class A
{
    int i = 0;
    void F() {
        i = 1;           // 错误，在声明之前使用了局部变量
        int i;
        i = 2;
    }
    void G() {
        int j = (j = 1); // 有效
    }
    void H() {
        int a = 1, b = ++a; // 有效
    }
}
```

在上面的方法 F 中，对 *i* 第一次赋值时，*i* 一定不是指在外部范围声明的字段 *i*；相反，它所引用的是局部变量 *i*，这会导致编译时错误，因为它在文本上位于该变量的声明之前。在方法 G 中，在 *j* 的声明初始值设定项中使用 *j* 是有效的，因为并未在局部变量声明符之前使用 *j*。

在方法 H 中，在同一局部变量声明内，后面的局部变量声明符正确引用前面的局部变量声明符中声明的局部变量。

局部变量的范围规则旨在保证表达式上、下文中使用的名称其含义在块中总是相同。如果局部变量的范围仅从它的声明扩展到块的结尾，则在上面的示例中，第一次赋值将分配给实例变量，第二次赋值将分配给局部变量。如果后来重新排列块的语句，则可能会导致编译时错误。

块中名称的含义可能因该名称的使用上、下文而异。在下面的示例中，名称 A 在表达式上、下文中用来引用局部变量 A，在类型上、下文中用来引用类 A。

```
using System;
class A {}
class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;           // 表达式上下文
        Type t = typeof(A);      // 类型上下文
        Console.WriteLine(s);    // 写出 "hello, world"
        Console.WriteLine(t);    // 写出 "A"
    }
}
```

实体的范围通常比该实体的声明空间包含更多的程序文本。具体说来，实体的范围可能包含一些声明，它们会引入一些新的声明空间，其中可能含有与该实体同名的新实体。这类声明导致原始的实体变为**隐藏的 (hidden)**。相反，当实体不是隐藏的时，就说它是**可见的 (visible)**。

当范围之间相重叠（或通过嵌套重叠，或通过继承重叠）时会发生名称隐藏。以下各

节介绍这两种隐藏类型的特性。

3.7.1 通过嵌套隐藏

以下各项活动会导致通过嵌套的名称隐藏发生：在命名空间内嵌套其他命名空间或类型；在类或结构中的嵌套类型；声明参数和局部变量。

下面的示例，在方法 F 中，实例变量 *i* 被局部变量 *i* 隐藏，但在方法 G 中，*i* 仍引用该实例变量。

```
class A
{
    int i = 0;
    void F() {
        int i = 1;
    }
    void G() {
        i = 1;
    }
}
```

当内部范围中的名称隐藏外部范围中的名称时，它隐藏该名称的所有重载匹配项。在下面的示例中，由于 F 的所有外部匹配项都被内部声明隐藏，因此调用 F(1) 调用在 Inner 中声明的 F。由于同样的原因，调用 F("Hello") 导致编译时错误。

```
class Outer
{
    static void F(int i) {}
    static void F(string s) {}
    class Inner
    {
        void G() {
            F(1);           // 调用 Outer.Inner.F
            F("Hello");     // 错误
        }
        static void F(long l) {}
    }
}
```

3.7.2 通过继承隐藏

当类或结构重新声明从基类继承的名称时，会发生通过继承的名称隐藏。这种类型的名称隐藏采取下列形式之一：

- 类或结构中引入的常数、字段、属性、事件或类型会把其基类中所有同名的成员隐藏起来。
- 类或结构中引入的方法隐藏所有同名的非方法基类成员，以及所有具有相同签名（方法名称和参数个数、修饰符和类型）的基类方法。
- 类或结构中引入的索引器隐藏所有具有相同签名（参数个数和类型）的基类索引器。

管理运算符声明（§ 10.9）的规则使派生类不可能声明与基类中的运算符具有相同签名的运算符。因此，运算符从不相互隐藏。

与隐藏外部范围中的名称相反，隐藏继承范围中的可访问名称会导致发出警告。在下

面的示例中，**Derived** 中的 **F** 声明导致报告一个警告。

```
class Base
{
    public void F() {}
}
class Derived: Base
{
    public void F() {}    // 警告，隐藏继承的名称
}
```

准确地说，隐藏继承的名称不是一个错误，因为这会限制基类按自身情况进行改进。例如，由于更高版本的 **Base** 引入了该类的早期版本中不存在的 **F** 方法，可能会发生上述情况。如果上述情况是一个错误，则当基类属于单独进行版本控制的类库时，对该基类的“任何”更改都有可能导致它的派生类变得无效。

通过使用 **new** 修饰符可以消除因隐藏继承的名称导致的警告：

```
class Base
{
    public void F() {}
}
class Derived: Base
{
    new public void F() {}
}
```

new 修饰符指示 **Derived** 中的 **F** 是“新的”，并且确实是有意隐藏继承成员。在声明新成员时，仅在该新成员的范围内隐藏被继承的成员。

```
class Base
{
    public static void F() {}
}
class Derived: Base
{
    new private static void F() {}    // 在 Derived 中隐藏 Base.F
}
class MoreDerived: Derived
{
    static void G() {F(); }           // 调用 Base.F
}
```

在上面的示例中，**Derived** 中的 **F** 声明隐藏从 **Base** 继承的 **F**，但由于 **Derived** 中的新 **F** 具有私有访问，它的范围不扩展到 **MoreDerived**，因此，**MoreDerived.G** 中的调用 **F()** 是有效的并将调用 **Base.F**。

3.8 命名空间和类型名称

C# 程序中的若干上下文要求指定命名空间名称或类型名称。两种形式的名称都写为以“.”标记分隔的一个或多个标识符。

namespace-name: (命名空间名称:)

namespace-or-type-name (命名空间或类型名称)

type-name: (类型名:)

namespace-or-type-name (命名空间或类型名称)

namespace-or-type-name: (命名空间或类型名称:)

identifier (标识符)

namespace-or-type-name . identifier (命名空间或类型名称 . 标识符)

“类型名”是一个“命名空间或类型名称”，它引用一个类型。需遵循下述的决策：“类型名”的“命名空间或类型名称”必须引用一个类型，否则将发生编译时错误。

“命名空间名称”是一个“命名空间或类型名称”，它引用一个命名空间。需遵循下述的决策：“命名空间名称”的“命名空间或类型名称”必须引用一个命名空间，否则将发生编译时错误。

“命名空间或类型名称”的含义按下述步骤确定。

- 如果“命名空间或类型名称”由单个标识符组成：

- ◆ 如果“命名空间或类型名称”出现在类或结构声明体内，则从该类或结构声明开始查找，遍及每个封闭它的类或结构声明（若它们存在的话），如果具有给定名称的成员存在、可访问且表示类型，则“命名空间或类型名称”引用该成员。请注意，当确定“命名空间或类型名称”的含义时，会忽略非类型成员（常数、字段、方法、属性、索引器、运算符、实例构造函数、析构函数和静态构造函数）。

- ◆ 否则，从发生“命名空间或类型名称”的命名空间开始，遍及每个封闭它的命名空间（若它们存在的话），直至全局命名空间结束，对下列步骤进行评估，直到找到实体。

- 如果命名空间包含具有给定名称的命名空间成员，则“命名空间或类型名称”引用该成员，并根据该成员归为命名空间或类型类别。

- 否则，如果命名空间有一个对应的命名空间声明，且“命名空间或类型名称”出现的位置包含在该命名空间声明中，则：

- 如果该命名空间声明包含一个将给定名称与一个导入的命名空间或类型关联的 `using` 别名指令，则“命名空间或类型名称”引用该命名空间或类型。

- 否则，如果该命名空间声明中有一个“`using` 命名空间指令”，它导入的那个命名空间内含有一个与给定名称完全匹配的类型，则“命名空间或类型名称”引用该类型。

- 否则，如果该“`using` 命名空间指令”导入的命名空间包含多个具有给定名称的类型，则“命名空间或类型名称”被认为是含义不清的，将导致发生错误。

- ◆ 否则，“命名空间或类型名称”就被认为是未定义的，导致发生编译时错误。

- 否则，“命名空间或类型名称”的形式为 `N.I`，其中 `N` 是由除最右边的标识符以外的所有标识符组成的“命名空间或类型名称”，`I` 是最右边的标识符。`N` 最先按“命名空间或类型名称”解析。如果对 `N` 的解析不成功，则发生编译时错误。

否则，N.I 按下面内容解析。

- ◆ 如果 N 是一个命名空间而 I 是该命名空间中可访问成员的名称，则 N.I 引用该成员，并根据该成员归为命名空间或类型类别。
- ◆ 如果 N 是类或结构类型而 I 是 N 中可访问类型的名称，则 N.I 引用该类型。
- ◆ 否则，N.I 是无效的命名空间或类型名称并将发生编译时错误。

每个命名空间和类型都具有一个完全限定名，该名称在所有其他命名空间或类型中惟一标识该命名空间或类型。命名空间或类型 N 的完全限定名按下面这样确定：

- 如果 N 是全局命名空间的成员，则它的完全限定名为 N。
- 否则，它的完全限定名为 S.N，其中 S 是声明了 N 的命名空间或类型的完全限定名。

换言之，N 的完全限定名是从全局命名空间开始通向 N 的标识符的完整分层路径。由于命名空间或类型的每个成员都必须具有惟一的名称，因此，如果将这些成员名称置于命名空间或类型的完全限定名之后，这样构成的成员完全限定名一定符合惟一性。

下面的示例显示了若干命名空间和类型声明及其关联的完全限定名。

```
class A {}           // A
namespace X          // X
{
    class B          // X.B
    {
        class C {}   // X.B.C
    }
    namespace Y       // X.Y
    {
        class D {}   // X.Y.D
    }
}
namespace X.Y         // X.Y
{
    class E {}        // X.Y.E
}
```

3.9 自动内存管理

C# 使用自动内存管理。它使开发人员不再需要以手动方式分配和释放由对象占用的内存。自动内存管理策略由垃圾回收器实现。对象的内存管理生存周期如下所示：

- 当创建对象时，将为其分配内存，运行构造函数，该对象被视为活对象。
- 如果该对象或它的任何部分在后续执行过程中不能被访问了（除了运行它的析构函数），则该对象被视为不再被使用，可以销毁。C# 编译器和垃圾回收器可以通过分析代码，确定哪些对象引用可能在将来被使用。例如，如果范围内的某个局部变量是现有的关于此对象的惟一的引用，但在当前执行点之后的任何后续执行过程中，该局部变量都不能被引用，那么垃圾回收器可以（但不是必须）认为该对象不再被使用。
- 一旦对象符合销毁条件，在稍后某个时间将运行该对象的析构函数（§ 10.12）（如

果有的话)。除非被显式调用所重写, 否则对象的析构函数只运行一次。

- 一旦运行对象的析构函数, 如果该对象或它的任何部分无法由任何可能的执行继续(包括运行析构函数)访问, 则该对象被视为不可访问, 可以回收。
- 最后, 在对象变得符合回收条件后, 垃圾回收器将释放与该对象关联的内存。

垃圾回收器维护对象的使用信息, 并利用此信息做出内存管理决定, 例如在内存中的何处安排一个新创建的对象、何时重定位对象, 以及对象何时不再被使用或不可访问。

与其他假定存在垃圾回收器的语言一样, C# 也被设计为使垃圾回收器可以实现广泛的内存管理策略。例如, C# 并不要求一定要运行析构函数, 不要求对象一符合条件就被回收, 也不要求析构函数以任何特定的顺序或在任何特定的线程上运行。

垃圾回收器的行为在某种程度上可通过类 `System.GC` 的静态方法来控制。该类可用于请求执行一次回收操作、运行(或不运行)析构函数, 等等。

由于垃圾回收器在决定何时回收对象和运行析构函数方面可以有很大的选择范围, 因此它的一个符合条件的实现所产生的输出可能与下面的代码所显示的不同。

```
using System;
class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}
class B
{
    object Ref;
    public B(object o) {
        Ref = o;
    }
    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}
class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

为创建类 A 的一个实例和类 B 的一个实例。当给变量 *b* 赋值 `null` 后, 这些对象变得符合垃圾回收条件, 这是因为从此往后, 任何用户编写的代码不可能再访问这些对象。输出可以为

```
Destruct instance of A
Destruct instance of B
```

或

```
Destruct instance of B
Destruct instance of A
```

这是因为该语言对于对象的垃圾回收顺序没有强加约束。

在微妙的情况中，“符合销毁条件”和“符合回收条件”之间的区别非常重要。例如：

```
using System;
class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}
class B
{
    public A Ref;
    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}
class Test
{
    public static A RefA;
    public static B RefB;
    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;
        // A 和 B 现在符合销毁条件
        GC.Collect();
        GC.WaitForPendingFinalizers();
        //现在 B 符合回收条件, 但 A 不符合
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}
```

在上面的程序中，如果垃圾回收器选择在 B 的析构函数之前运行 A 的析构函数，则该程序的输出可能是：

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```

请注意，虽然 A 的实例被当做“没有被使用”，并且 A 的析构函数已被运行过了，但仍可能从其他析构函数调用 A 的方法（此例中是指 F）。还请注意，运行析构函数可能导致对象再次从主干程序中变得可用。在此例中，运行 B 的析构函数导致了先前没有被使用的 A 的实例变得可从当前有效地引用 Test.RefA 访问。调用 WaitForPendingFinalizers 后，B 的实例符合回收条件，但由于引用 Test.RefA 的缘故，A 的实例不符合回收条件。

为了避免混淆和意外的行为，好的做法通常是让析构函数只对存储在它们的对象本身字段中的数据执行清理，而不对它所引用的其他对象或静态字段执行任何操作。

3.10 执行顺序

C# 程序执行时，在临界执行点保留每个执行线程的副作用。按照定义，副作用（**side effect**）是指读写易失性字段、写入非易失性变量、写入外部资源和引发异常。临界执行点（这些副作用的顺序必须保存在其中）是指下列各活动：引用一些易失性字段（§ 10.4.3）；引用 `lock` 语句（§ 8.12）；引用线程的创建与终止。执行环境可以随便更改 C# 程序的执行顺序，但受下列约束限制：

- 在执行线程中需保持数据依赖性。就是说，在计算每个变量的值时，就好像线程中的所有语句都是按原始程序顺序执行的。
- 保留初始化的排序规则（§ 10.4.4 和 § 10.4.5）。
- 对于易失性读写（§ 10.4.3）的副作用的顺序需保持不变。此外，如果能推断出表达式的值是“不会被使用的”而且不会产生有效的副作用（包括由调用方法或访问易失性字段导致的任何副作用），则执行环境甚至不需要计算一个表达式的各个部分。当程序执行被异步事件（例如其他线程引发的异常）中断时，它不保证可观察到的副作用以原有的程序顺序出现。

第4章 类型

C# 语言的类型划分为两大类：值类型和引用类型。

type: (类型:)

value-type (值类型)

reference-type (引用类型)

第三种类型是指针，只能用在不安全代码中，§ 18.2 对此做了进一步的探讨。

值类型与引用类型的不同之处在于：值类型的变量直接包含其数据，而引用类型的变量存储对其数据的引用，后者称为对象。对于引用类型，两个变量可能引用同一个对象，因此对一个变量的操作可能影响另一个变量所引用的对象。对于值类型，每个变量都有自己的数据副本，对一个变量的操作不可能影响另一个变量。

C# 的类型系统是统一的，因此任何类型的值都可以按对象处理。C# 中的每个类型直接或间接地从 object 类类型派生，而 object 是所有类型的最终基类。引用类型的值都被当做对象来处理，这是因为这些值可以简单地被视为属于 object 类型。值类型的值则通过执行装箱和取消装箱操作 (§ 4.3) 按对象处理。

4.1 值类型

值类型或者是结构类型，或者是枚举类型。C# 提供一个称为“简单类型”的预定义的结构类型集。简单类型通过保留字来标识。

value-type: (值类型:)

struct-type (结构类型)

enum-type (枚举类型)

struct-type: (结构类型:)

type-name (类型名称)

simple-type (简单类型)

simple-type: (简单类型:)

numeric-type (数值类型)

bool

numeric-type: (数值类型:)

integral-type (整型)

floating-point-type (浮点类型)

decimal

integral-type: (整型:)

```
sbyte
byte
short
ushort
int
uint
long
ulong
char
```

floating-point-type: (浮点类型:)

```
float
double
```

enum-type: (枚举类型:)

type-name (类型名)

值类型的变量总是包含该类型的值。与引用类型不同，值类型的值不可能为 `null`，也不可能引用派生程度较大的类型的对象。

值类型的变量赋值会创建所赋的值的一个副本。这不同于引用类型的变量赋值，引用类型的变量赋值复制的是引用而不是由引用标识的对象。

4.1.1 System.ValueType 类型

所有值类型从类 `System.ValueType` 隐式继承，后者又从类 `object` 继承。任何类型都不可能从值类型派生，因此，所有值类型都是隐式密封的 (§ 10.1.1.2)。

注意，`System.ValueType` 本身不是值类型。相反，它属于类类型，所有值类型都从它自动派生。

4.1.2 默认构造函数

所有值类型都隐式声明一个称为**默认构造函数** (**default constructor**) 的公共无参数实例构造函数。默认构造函数返回一个零初始化实例，它就是该值类型的**默认值** (**default value**)。

- 对于所有简单类型，默认值是将其所有位都置零的位模式所形成的值：
 - 对于 `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` 和 `ulong`, 默认值为 0。
 - 对于 `char`, 默认值为 `'\x0000'`。
 - 对于 `float`, 默认值为 `0.0f`。
 - 对于 `double`, 默认值为 `0.0d`。
 - 对于 `decimal`, 默认值为 `0.0m`。
 - 对于 `bool`, 默认值为 `false`。
- 对于枚举类型 `E`, 默认值为 0。

- 对于结构类型，默认值是通过将所有值类型字段设置为它们的默认值、将所有引用类型字段设置为 `null` 而产生的值。

与任何其他实例构造函数一样，值类型的默认构造函数也是用 `new` 运算符调用的。出于效率原因，实际上，不必故意调用它的构造函数。在下面的示例中，变量 `i` 和 `j` 都被初始化为零。

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}
```

由于每个值类型都隐式地具有一个公共无参数实例构造函数，因此，一个结构类型中不可能包含一个关于无参数构造函数的显式声明。但允许结构类型声明参数化实例构造函数（§ 11.3.8）。

4.1.3 结构类型

结构类型是一种值类型，它可以声明常数、字段、方法、属性、索引器、运算符、实例构造函数、静态构造函数和嵌套类型。有关结构类型的介绍详见 § 11。

4.1.4 简单类型

C# 提供称为简单类型的预定义结构类型集。简单类型通过保留字来标识，而这些保留字只是 `System` 命名空间中预定义的结构类型的别名，详见表 4.1。

表 4.1 保留字与结构类型

保留字	化名的类型
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

由于简单类型是结构类型的别名，因此每个简单类型都具有成员。例如，`int` 具有在

`System.Int32` 中声明的成员及从 `System.Object` 继承的成员，允许使用下面的语句：

```
int i = int.MaxValue;           // System.Int32.MaxValue 常数
string s = i.ToString();        // System.Int32.ToString() 实例方法
string t = 123.ToString();      // System.Int32.ToString() 实例方法
```

简单类型与其他结构类型的不同之处在于：简单类型允许某些附加的操作。

- 大多数简单类型允许通过编写文本（§ 2.4.4）来创建值。例如，123 是 `int` 类型的文本，'a' 是 `char` 类型的文本。C# 没有普遍地为结构类型设置类似的以文本创建值的规则，所以其他结构类型的非默认值最终总是通过这些结构类型的实例构造函数来创建的。
- 当表达式的操作数都是简单类型常数时，编译器可以在编译时计算表达式。这样的表达式称为常数表达式（§ 7.15）。涉及其他结构类型所定义的运算符的表达式不被视为常数表达式。
- 通过 `const` 声明可以声明简单类型的常数（§ 10.3）。常数不可能属于其他结构类型，但 `static readonly` 字段提供了类似的效果。
- 涉及简单类型的转换可以参与由其他结构类型定义的转换运算符的计算，但用户定义的转换运算符从来不能参与其他用户定义运算符的计算（§ 6.4.2）。

4.1.5 整型

C# 支持 9 种整型：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong` 和 `char`。整型具有以下所列的大小和取值范围：

- `sbyte` 类型表示有符号 8 位整数，其值介于 -128 和 127 之间。
- `byte` 类型表示无符号 8 位整数，其值介于 0 和 255 之间。
- `short` 类型表示有符号的 16 位整数，其值介于 -32 768 和 32 767 之间。
- `ushort` 类型表示无符号的 16 位整数，其值介于 0 和 65 535 之间。
- `int` 类型表示有符号 32 位整数，其值介于 -2 147 483 648 和 2 147 483 647 之间。
- `uint` 类型表示无符号 32 位整数，其值介于 0 和 4 294 967 295 之间。
- `long` 类型表示有符号的 64 位整数，其值介于 -9 223 372 036 854 775 808 和 9 223 372 036 854 775 807 之间。
- `ulong` 类型表示无符号的 64 位整数，其值介于 0 和 18 446 744 073 709 551 615 之间。
- `char` 类型表示无符号 16 位整数，其值介于 0 和 65 535 之间。`char` 类型的可能值集与 Unicode 字符集相对应。虽然 `char` 的表示形式与 `ushort` 相同，但一种类型上允许实施的所有操作并非都可以用在另一种类型上。

整型一元运算符和二元运算符总是对符号 32 位精度、无符号的 32 位精度、有符号 64 位精度或无符号 64 位精度进行操作。

- 对于一元运算符 `+` 和 `~`，操作数转换为 `T` 类型，其中 `T` 是 `int`、`uint`、`long` 和 `ulong` 中第一个可以完全表示操作数的所有可能值的类型。然后用 `T` 类型的精度执行运算，结果的类型是 `T` 类型。

- 对于一元运算符 `-`，操作数转换为类型 `T`，其中 `T` 是 `int` 和 `long` 中第一个可以完全表示操作数的所有可能值的类型。然后用 `T` 类型的精度执行运算，结果的类型是 `T` 类型。一元运算符 `-` 不能应用于类型 `ulong` 的操作数。
- 对于 `+`，`-`，`*`，`/`，`%`，`&`，`^`，`|`，`==`，`!=`，`>`，`<`，`>=` 和 `<=` 二元运算符，操作数转换为类型 `T`，其中 `T` 是 `int`，`uint`，`long` 和 `ulong` 中第一个可以完全表示两个操作数的所有可能值的类型。然后用 `T` 类型的精度执行运算，运算的结果的类型也属于 `T`（对于关系运算符为 `bool`）。对于二元运算符，不允许一个操作数为 `long` 类型而另一个操作数为 `ulong` 类型。
- 对于二元运算符 `<<` 和 `>>`，左操作数转换为 `T` 类型，其中 `T` 是 `int`，`uint`，`long` 和 `ulong` 中第一个可以完全表示操作数的所有可能值的类型。然后用 `T` 类型的精度执行运算，结果的类型是 `T` 类型。

`char` 类型按分类归属为整型类别，但它在以下两个方面不同于其他整型：

- 不存在从其他类型到 `char` 类型的隐式转换。具体说来，即使 `sbyte`，`byte` 和 `ushort` 类型具有完全可以用 `char` 类型来表示的值范围，也不存在从 `sbyte`，`byte` 或 `ushort` 到 `char` 的隐式转换。
- `char` 类型的常数必须写成字符或写成强制转换为类型 `char` 的整数。例如，`(char)10` 与 `'\x000A'` 是相同的。

`checked` 和 `unchecked` 运算符和语句用于控制整型算术运算和转换的溢出检查（§ 7.5.12）。在 `checked` 上下文中，溢出产生编译时错误或导致引发 `System.OverflowException`。在 `unchecked` 上下文中将忽略溢出，任何与目标类型不匹配的高序位都被放弃。

4.1.6 浮点型

C# 支持两种浮点型：`float` 和 `double`。`float` 和 `double` 类型用 32 位单精度和 64 位双精度 IEEE 754 格式来表示，这些格式提供以下几组值：

- 正零和负零。大多数情况下，正零和负零的行为与简单的值零相同，但某些运算会区别对待此两种零（§ 7.7.2）。
- 正无穷大和负无穷大。无穷大是由非零数字被零除这样的运算产生的。例如，`1.0 / 0.0` 产生正无穷大，而 `-1.0 / 0.0` 产生负无穷大。
- 非数字值（**Not-a-Number**），常缩写为 `NaN`。`NaN` 是由无效的浮点运算（如零被零除）产生的。
- 以 $s \times m \times 2^e$ 形式表示的非零值的有限集合，其中 s 为 1 或 -1， m 和 e 由特定的浮点型确定：对于 `float` 类型，为 $0 < m < 2^{24}$ 和 $-149 \leq e \leq 104$ ，对于 `double` 类型，则为 $0 < m < 2^{53}$ 和 $-1075 \leq e \leq 970$ 。非标准化的浮点数被视为有效非零值。

`float` 类型可表示精度为 7 位、在大约 $1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$ 的范围内的值。

`double` 类型可表示精度为 15 位或 16 位、在大约 $5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$ 的范围内的值。

如果二元运算符的一个操作数为浮点型，则另一个操作数必须为整型或浮点型，并且运算按下面这样计算：

- 如果一个操作数为整型，则该操作数转换为与另一个操作数的类型相同的浮点型。
- 然后，如果任一操作数的类型为 `double`，则另一个操作数转换为 `double`。至少用 `double` 范围和精度执行运算，结果的类型为 `double`（对于关系运算符则为 `bool`）。
- 否则，至少用 `float` 范围和精度执行运算，结果的类型为 `float`（对于关系运算符则为 `bool`）。

浮点运算符（包括赋值运算符）从来不产生异常。相反，在异常情况下，浮点运算产生零、无穷大或 NaN，如下所述：

- 如果浮点运算的结果对于目标格式太小，则运算结果变成正零或负零。
- 如果浮点运算的结果对于目标格式太大，则运算结果变成正无穷大或负无穷大。
- 如果浮点运算无效，则运算的结果变成 NaN。
- 如果浮点运算至少一个操作数为 NaN，则运算的结果变成 NaN。

可以用比运算的结果类型更高的精度来执行浮点运算。例如，某些硬件结构支持比 `double` 类型具有更大的范围和精度的“extended”或“long double”浮点型，并隐式地使用这种更高精度的类型执行所有浮点运算。只有以额外的性能开销为代价，才能使这样的硬件结构用“较低”的精度执行浮点运算。C# 采取的是允许将更高的精度类型用于所有浮点运算，而不是强制执行规定的精度，造成同时损失性能和精度。除了传递更精确的结果外，这样做很少会产生任何可察觉的效果。但是，在 $x * y / z$ 形式的表达式中，如果其中的乘法产生超出 `double` 范围的结果，而后面的除法使临时结果返回到 `double` 范围内，则以更大范围的格式去计算该表达式时，可能会产生有限值的结果（本来应是无穷大）。

4.1.7 decimal 类型

`decimal` 类型是适合财务和货币计算的 128 位数据类型。`decimal` 类型可以表示具有 28 或 29 个有效数字、从 1.0×10^{-28} 到大约 7.9×10^{28} 范围内的值。

`decimal` 类型的有限值集的形式为 $(-1)^s \times c \times 10^{-e}$ ，其中符号 s 是 0 或 1，系数 c 由 $0 \leq c < 2^{96}$ 给定，小数位数 e 满足 $0 \leq e \leq 28$ 。`decimal` 类型不支持有符号的零、无穷大或 NaN。`decimal` 可用一个 96 位整数配上以 10 的幂标定的单位（定位小数点）来表示。对于绝对值小于 1.0m 的 `decimal`，它的值最多精确到第 28 位小数。对于绝对值大于或等于 1.0m 的 `decimal`，它的值精确到 28 或 29 个有效数字。与 `float` 和 `double` 数据类型相反，十进制小数数字（如 0.1）可以精确地用 `decimal` 表示形式来表示。在 `float` 和 `double` 表示形式中，这类数字通常变成无限小数，使这些表示形式更容易发生舍入错误。

如果二元运算符的一个操作数为 `decimal` 类型，则另一个操作数必须为整型或 `decimal` 类型。如果存在一个整型操作数，则它要在执行运算前转换为 `decimal`。

`decimal` 类型值的运算结果是这样得出的：先计算一个精确结果（按每个运算符的定

义保留小数位数), 然后舍入以适合表示形式。结果舍入到最接近的可表示值, 当结果同样地接近于两个可表示值时, 舍入到最小有效位数位置中为偶数的值 (这称为“银行家舍入法”)。零结果总是包含符号 0 和小数位数 0。

如果十进制算术运算产生一个绝对值小于或等于 5×10^{-29} 的值, 则运算结果变为零。如果 decimal 算术运算产生的值对于 decimal 格式太大, 则将引发 System.OverflowException。

与浮点型相比, decimal 类型具有较高的精度, 但取值范围较小。因此, 从浮点型到 decimal 的转换可能会产生溢出异常, 而从 decimal 到浮点型的转换则可能导致精度损失。由于这些原因, 在浮点型和 decimal 之间不存在隐式转换, 如果没有显式地标出强制转换, 就不可能在同一表达式中同时使用浮点操作数和 decimal 操作数。

4.1.8 bool 类型

bool 类型表示布尔逻辑量。bool 类型的可能值为 true 和 false。

在 bool 类型和其他类型之间不存在标准转换。具体说来, bool 类型与整型截然不同, 不能用 bool 值代替整数值, 反之亦然。

在 C 语言和 C++ 语言中, 零整数或浮点值或空指针可以转换为布尔值 false, 非零整数或浮点值或非空指针可以转换为布尔值 true。在 C# 中, 这种转换是通过显式地将整数或浮点值与零进行比较, 或者显式地将对象引用与 null 进行比较来完成的。

4.1.9 枚举类型

枚举类型是具有命名常数的独特的类型。每个枚举类型都具有一个基础类型, 这必须是 byte, sbyte, short, ushort, int, uint, long 或 ulong。枚举类型的值集和它的基础类型的值集相同。枚举类型的值并不只限于那些命名常数的值。枚举类型是通过枚举声明 (§ 14.1) 定义的。

4.2 引用类型

引用类型是类类型、接口类型、数组类型或委托类型。

reference-type: (引用类型:)

class-type (类类型)

interface-type (接口类型)

array-type (数组类型)

delegate-type (委托类型)

class-type: (类类型:)

type-name (类型名)

object

string
interface-type: (接口类型:)
 type-name (类型名)
array-type: (数组类型:)
 non-array-type rank-specifiers (非数组类型 秩说明符)
non-array-type: (非数组类型:)
type (类型)
rank-specifiers: (秩说明符:)
 rank-specifier (秩说明符)
 rank-specifiers rank-specifier (秩说明符 秩说明符)

rank-specifier: (秩说明符:)
[dim-separators_{opt}] ([维度分隔符_{可选}])
dim-separators: (维度分隔符:)
 ,
 dim-separators , (维度分隔符 ,)
delegate-type: (委托类型:)
 type-name (类型名)

引用类型值是对该类型的某个实例的一个引用，后者称为对象。`null` 值比较特别，它适用于所有引用类型，用来表示“没有被引用的实例”。

4.2.1 类类型

类类型定义了包含数据成员、函数成员和嵌套类型的数据结构，其中数据成员包括常数和字段，函数成员包括方法、属性、事件、索引器、运算符、实例构造函数、析构函数和静态构造函数。类类型支持继承，继承是派生类可用来扩展和专门化基类的一种机制。类类型的实例是用对象创建表达式 (§ 7.5.10.1) 创建的。

有关类类型的介绍详见 § 10。
某些预定义类类型在 C# 语言中有特殊含义，如表 4.2 所示。

表 4.2 C# 中预定义类类型的特殊含义

类 类 型	说 明
System.Object	所有其他类型的最终基类 (§ 4.2.2)
System.String	C# 语言的字符串类型 (§ 4.2.3)
System.ValueType	所有值类型的基类 (§ 4.1.1)
System.Enum	所有枚举类型的基类 (§ 14)
System.Array	所有数组类型的基类 (§ 12)
System.Delegate	所有委托类型的基类 (§ 15)
System.Exception	所有异常类型的基类 (§ 16)

4.2.2 对象类型

`object` 类类型是所有其他类型的最终基类。C# 中的每种类型都是直接或间接从 `object` 类类型派生的。

关键字 `object` 只是预定义类 `System.Object` 的别名。

4.2.3 string 类型

`string` 类型是直接从 `object` 继承的密封类类型。`string` 类的实例表示 Unicode 字符串。

`string` 类型的值可以写为字符串 (§ 2.4.4)。

关键字 `string` 只是预定义类 `System.String` 的别名。

4.2.4 接口类型

一个接口定义一个协定。实现接口的类或结构必须遵守其协定。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。

有关接口类型的介绍详见 § 13。

4.2.5 数组类型

数组是一种数据结构，它包含可通过计算索引访问的零个或多个变量。数组中包含的变量（又称数组的元素）具有相同的类型，该类型称为数组的元素类型。

有关数组类型的介绍详见 § 12。

4.2.6 委托类型

委托是一种数据结构，它引用一个或多个方法，对于实例方法，还引用这些方法所对应的对象实例。

在 C 或 C++ 中与委托最接近的是函数指针，但函数指针只能引用静态函数，而委托则既可以引用静态方法，也可以引用实例方法。在引用实例方法中，委托不仅存储了一个对该方法入口点的引用，还存储了一个对相应的对象实例的引用，该方法就是通过此对象实例被调用的。

有关委托类型的介绍详见 § 15。

4.3 装箱和取消装箱

装箱和取消装箱的概念是 C# 的类型系统的核心。它在“值类型”和“引用类型”之间架起了一座桥梁，使得任何“值类型”的值都可以转换为 `object` 类型的值，反过来转换也可以。装箱和取消装箱使我们能够统一地来考察类型系统，其中任何类型的值最终都可以按对象处理。

4.3.1 装箱转换

装箱转换允许将“值类型”隐式转换为“引用类型”。存在下列装箱转换：

- 从任何“值类型”（包括任何“枚举类型”）到类型 `object`。
- 从任何“值类型”（包括任何“枚举类型”）到类型 `System.ValueType`。
- 从任何“值类型”到“值类型”实现的任何“接口类型”。
- 从任何“枚举类型”到 `System.Enum` 类型。

将“值类型”的值装箱的操作包括：分配一个对象实例并将“值类型”的值复制到该实例中。

最能说明“值类型”的值的实际装箱过程的办法是，设想有一个为该类型设置的装箱类。对任何“值类型”的 `T` 而言，装箱类的行为可用下列声明来描述：

```
sealed class T_Box: System.ValueType
{
    T value;
    public T_Box(T t) {
        value = t;
    }
}
```

`T` 类型值 `v` 的装箱过程现在包括了执行表达式 `new T_Box(v)`，以及将结果实例作为 `object` 类型的值返回。因此，下面的语句

```
int i = 123;
object box = i;
```

在概念上相当于

```
int i = 123;
object box = new int_Box(i);
```

实际上，像前面的 `T_Box` 和 `int_Box` 这样的装箱类并不存在，并且装了箱的值的动态类型也不会真的属于类类型。相反，`T` 类型的装了箱的值属于动态类型 `T`，若用 `is` 运算符来检查动态类型的话，也仅能引用类型 `T`。例如：

```
int i = 123;
object box = i;
if (box is int) {
    Console.WriteLine("Box contains an int");
}
```

将在控制台上输出字符串“Box contains an int”。

装箱转换隐含着复制一份待装箱的值。这不同于从引用类型到 `object` 类型的转换，在后一种转换中，转换后的值继续引用同一实例，只是将它当做派生程度较小的 `object` 类型而已。例如，设有下列的声明

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

则下面的语句

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.WriteLine(((Point)box).x);
```

将在控制台上输出值 10，因为将 `p` 赋值给 `box` 是一个隐式的装箱操作，它将复制 `p` 的值。如果将 `Point` 声明为 `class`，由于 `p` 和 `box` 将引用同一个实例，因此输出值为 20。

4.3.2 取消装箱转换

取消装箱转换允许将引用类型显式转换为值类型。存在以下取消装箱转换：

- 从类型 `object` 到任何值类型（包括任何枚举类型）。
- 从类型 `System.ValueType` 到任何值类型（包括任何枚举类型）。
- 从任何接口类型到实现了该接口类型的任何值类型。
- 从 `System.Enum` 类型到任何枚举类型。

取消装箱操作包括以下两个步骤：首先检查该对象实例是否是某个给定的值类型的装了箱的值，然后将值从实例中复制出来。

参照前一节中关于假想的装箱类的描述，从对象 `box` 到值类型 `T` 的取消装箱转换相当于执行表达式 `((T_Box)box).value`。因此，下面的语句

```
object box = 123;
int i = (int)box;
```

在概念上相当于

```
object box = new int_Box(123);
int i = ((int_Box)box).value;
```

为使到给定值类型的取消装箱转换在运行时取得成功，源操作数的值必须是对某个对象的引用，而该对象先前是通过将该值类型的某个值装箱而创建的。如果源操作数为 `null`，则将引发 `System.NullReferenceException`。如果源操作数是对不兼容对象的引用，则将引发 `System.InvalidCastException`。

第 5 章 变量

变量表示存储位置。每个变量都具有一个类型，它确定哪些值可以存储在该变量中。C# 是一种类型安全的语言，C# 编译器保证存储在变量中的值总是具有合适的类型。通过赋值或使用 ++ 和 -- 运算符可以更改变量的值。

在可以获取变量的值之前，变量必须已明确赋值（§ 5.3）。

如下面的章节所述，变量是初始已赋值或初始未赋值。初始已赋值的变量有一个正确定义的初始值，并且总是被视为已明确赋值。初始未赋值的变量没有初始值。为了使初始未赋值的变量在某个位置被视为已明确赋值，变量赋值必须发生在通向该位置的每个可能的执行路径中。

5.1 变量类别

C# 定义了 7 种变量类别：静态变量、实例变量、数组元素、值参数、引用参数、输出参数和局部变量。后面的章节将介绍这些类别。

下面是一个示例：

```
class A
{
    public static int x;
    int y;
    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

x 是静态变量，*y* 是实例变量，*v*[0] 是数组元素，*a* 是值参数，*b* 是引用参数，*c* 是输出参数，*i* 是局部变量。

5.1.1 静态变量

用 `static` 修饰符声明的字段称为静态变量。静态变量在包含了它的那个类型的静态构造函数（§ 10.11）执行之前就存在了，在关联的应用程序域终止时终止。

静态变量的初始值是该变量的类型的默认值（§ 5.2）。

出于明确赋值检查的目的，静态变量被视为初始已赋值。

5.1.2 实例变量

未用 `static` 修饰符声明的字段称为实例变量。

5.1.2.1 类中的实例变量

类的实例变量在创建该类的新实例时开始存在，在所有对该实例的引用都已终止，并且已执行了该实例的析构函数（若存在）时终止。

类实例变量的初始值是该变量的类型的默认值（§ 5.2）。

出于明确赋值检查的目的，类的实例变量被视为初始已赋值。

5.1.2.2 结构中的实例变量

结构的实例变量与它所属的结构变量具有完全相同的生存期。换言之，当结构类型的变量开始存在或停止存在时，该结构的实例变量也随之存在或消失。

结构的实例变量与包含它的结构变量具有相同的初始赋值状态。换言之，当结构变量本身被视为初始已赋值时，它的实例变量也被视为初始已赋值。而当结构变量被视为初始未赋值时，它的实例变量同样被视为未赋值。

5.1.3 数组元素

数组的元素在创建数组实例时开始存在，在没有对该数组实例的引用时停止存在。

每个数组元素的初始值都是其数组元素类型的默认值（§ 5.2）。

出于明确赋值检查的目的，数组元素被视为初始已赋值。

5.1.4 值参数

未用 `ref` 或 `out` 修饰符声明的参数为值参数。

值参数在调用该参数所属的函数成员（方法、实例构造函数、访问器或运算符）（§ 7.4）时开始存在，并用调用中给定的自变量的值初始化。当返回该函数成员时值参数停止存在。

出于明确赋值检查的目的，值参数被视为初始已赋值。

5.1.5 引用参数

用 `ref` 修饰符声明的参数是引用参数。

引用参数不创建新的存储位置，而是与那个在对该函数成员调用中被当做“自变量”的变量表示同一个存储位置。因此，引用参数的值总是与基础变量相同。

下面的明确赋值规则适用于引用参数。注意 § 5.1.6 中描述的输出参数的不同规则。

- 变量在可以作为引用参数在函数成员调用中传递之前，必须已明确赋值（§ 5.3）。
- 在函数成员内部，引用参数被视为初始已赋值。

在结构类型的实例方法或实例访问器内部，`this` 关键字的行为与该结构类型的引用参数完全相同（§ 7.5.7）。

5.1.6 输出参数

用 `out` 修饰符声明的参数是输出参数。

输出参数不创建新的存储位置，而是与那个在对该函数成员调用中被当做“自变量”的变量表示同一个存储位置。因此，输出参数的值总是与基础变量相同。

下面的明确赋值规则应用于输出参数。注意 § 5.1.5 中描述的引用参数的不同规则。

- 变量在可以作为输出参数在函数成员调用中传递之前不一定要明确赋值。
- 在正常完成函数成员调用之后，每个作为输出参数传递的变量都被认为在该执行路径中已赋值。
- 在函数成员内部，输出参数被视为初始未赋值。
- 函数成员的每个输出参数在该函数成员正常返回前都必须已明确赋值（§ 5.3）。

在结构类型的实例构造函数内部，`this` 关键字的行为与结构类型的输出参数完全相同（§ 7.5.7）。

5.1.7 局部变量

局部变量是通过局部变量声明来声明的，此声明可以出现在块、`for` 语句、`switch` 语句或 `using` 语句中。

局部变量的生存期是程序执行过程中的某一“段”，在此期间，一定会为该局部变量保留存储。此生存期从进入与它关联的块、`for` 语句、`switch` 语句或 `using` 语句开始，一直延续到对应的块、`for` 语句、`switch` 语句或 `using` 语句的执行以任何方式结束为止（进入封闭块或调用方法会挂起（但不会结束）当前的块、`for` 语句、`switch` 语句或 `using` 语句的执行）。如果以递归方式进入父块、`for` 语句、`switch` 语句或 `using` 语句，则每次都创建局部变量的新实例，并且重新计算它的局部变量初始值设定项（如果有的话）。

局部变量不自动初始化，因此没有默认值。出于明确赋值检查的目的，局部变量被视为初始未赋值。局部变量声明可包括局部变量初始值设定项，在此情况下变量被视为在它的整个范围内（局部变量初始值设定项中提供的表达式内除外）已明确赋值。

在局部变量的范围内，在局部变量声明符之前的文本位置引用该局部变量将导致编译时错误。

局部变量的实际生存期依赖于具体实现。例如，编译器可能静态地确定块中的某个局部变量只用于该块的一小部分。根据这种分析，编译器生成的代码可能会提前回收该变量的存储（相对于包含该变量的生存期）。

局部引用变量所引用的存储的回收（§ 3.9）与该局部引用变量的生存期无关。

`foreach` 语句和 `try` 语句的特定 `catch` 子句也声明局部变量。对于 `foreach` 语句，局部变量是一个迭代变量（§ 8.8.4）。对于特定的 `catch` 子句，局部变量是一个异常变量（§ 8.10）。`foreach` 语句或特定的 `catch` 子句所声明的局部变量被视为在它的整个范围内已明

确赋值。

5.2 默认值

以下类别的变量自动初始化为它们的默认值：

- 静态变量。
- 类实例的实例变量。
- 数组元素。

变量的默认值取决于该变量的类型，并按下面这样确定：

- 对于值类型的变量，默认值与该值类型的默认构造函数（§ 4.1.2）所计算的值相同。
- 对于“引用类型”的变量，默认值为 `null`。

初始化为默认值的实现方法一般是让内存管理器或垃圾回收器在分配内存以供使用之前，将内存初始化为“所有位归零”（all-bits-zero）。由于这个原因，使用所有位归零来表示空（`null`）引用很方便。

5.3 明确赋值

在函数成员可执行代码中的给定位置，如果编译器可通过静态流程分析证明变量已自动初始化或已成为至少一个赋值的目标，则称该变量已明确赋值。明确赋值的规则为：

- 初始已赋值变量（§ 5.3.1）总是被视为已明确赋值。
- 如果所有可能通向给定位置的执行路径都至少包含以下内容之一，则初始未赋值变量（§ 5.3.2）被视为在该位置已明确赋值。
 - 将变量作为左操作数的简单赋值（§ 7.13.1）。
 - 将变量作为输出参数传递的调用表达式（§ 7.5.5）或对象创建表达式（§ 7.5.10.1）。
 - 对于局部变量，包含变量初始值设定项的局部变量声明（§ 8.5.1）。

关于对结构类型变量的实例变量是否明确赋值，既可个别地也可作为整体进行跟踪。除了上述规则，下面的规则也应用于结构类型变量及其实例变量：

- 如果包含实例变量的那个结构类型变量被视为已明确赋值，则该实例变量被视为已明确赋值。
- 如果结构类型变量的每个实例变量都被视为已明确赋值，则该结构类型变量被视为已明确赋值。

在下列上下文中要求实施明确赋值：

- 变量必须在获取其值的每个位置都已明确赋值。这确保不会出现未定义的值。变量在表达式中出现则认为要获取该变量的值，除非以下之一成立：
 - 该变量为简单赋值的左操作数。
 - 该变量作为输出参数传递。

- 该变量为结构类型变量并作为成员访问的左操作数出现。
- 变量必须在它作为引用参数传递的每个位置都已明确赋值。这确保了被调用的函数成员可以将引用参数视为初始已赋值。
- 函数成员的所有输出参数必须在函数成员返回的每个位置都已明确赋值，返回位置包括通过 `return` 语句实现的返回，或者通过执行语句到达函数成员体结尾的返回。这确保了函数成员不在输出参数中返回未定义的值，从而使编译器能够把一个对函数成员的调用当做对某些变量的赋值，这些变量在该调用中被当做输出参数传递。
- 结构类型实例构造函数的 `this` 变量必须在该实例构造函数返回的每个位置明确赋值。

5.3.1 初始已赋值变量

以下类别的变量属于初始已赋值变量：

- 静态变量。
- 类实例的实例变量。
- 初始已赋值结构变量的实例变量。
- 数组元素。
- 值参数。
- 引用参数。
- 在 `catch` 子句或 `foreach` 语句中声明的变量。

5.3.2 初始未赋值变量

以下类别的变量属于初始未赋值变量：

- 初始未赋值结构变量的实例变量。
- 输出参数，包括结构实例构造函数的 `this` 变量。
- 局部变量，在 `catch` 子句或 `foreach` 语句中声明的那些除外。

5.3.3 确定明确赋值的细则

为了确定每个已使用变量都已明确赋值，编译器必须使用与本节中描述的进程等效的进程。

编译器处理每个具有一个或多个初始未赋值变量的函数成员体。对于每个初始未赋值的变量 `v`，编译器在函数成员中的下列每个点上确定 `v` 的**明确赋值状态**（**definite assignment state**）：

- 在每个语句的开头处。
- 在每个语句的结束点（§ 8.1）。
- 在每个将控制转移到另一个语句或语句结束点的 `arc` 上。

- 在每个表达式的开头处。
 - 在每个表达式的结尾处。
- v 的明确赋值状态可以是：
- 明确赋值。这表明在能达到该点的所有可能的控制流上， v 都已赋值。
 - 未明确赋值。当在 `bool` 类型表达式结尾处确定变量的状态时，未明确赋值的变量的状态可能（但不一定）属于下列子状态。
 - 在 `true` 表达式后明确赋值。此状态表明如果该布尔表达式计算为 `true`，则 v 是明确赋值的，但如果布尔表达式计算为 `false`，则不一定要赋值。
 - 在 `false` 表达式后明确赋值。此状态表明如果该布尔表达式计算为 `false`，则 v 是明确赋值的，但如果布尔表达式计算为 `true`，则不一定要赋值。

下列规则控制变量 v 的状态在每个位置是如何确定的。

5.3.3.1 一般语句规则

下面是适用于语句的一般规则。

- v 在函数成员体的开头处不是明确赋值的。
- v 在任何无法访问的语句的开头处都是明确赋值的。
- 在任何其他语句开头处，为了确定 v 的明确赋值状态，请检查以该语句开头处为目标的所有控制流转移上的 v 的明确赋值状态。当且仅当 v 在所有此类控制流转移上是明确赋值的时， v 才在该语句的开始处明确赋值。确定可能的控制流转移集的方法与检查语句可达性的方法（§ 8.1）相同。
- 在 `block`, `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using` 或 `switch` 等语句的结束点处，为了确定 v 的明确赋值状态，需检查以该语句结束点为目标的所有控制流转移上的 v 的明确赋值状态。如果 v 在所有此类控制流转移上是明确赋值的，则 v 在该语句结束点明确赋值。否则， v 在语句结束点处不是明确赋值的。确定可能的控制流转移集的方法与检查语句可达性的方法（§ 8.1）相同。

5.3.3.2 块语句、checked 语句和 unchecked 语句

在指向位于某块中语句列表的第一个语句（如果语句列表为空，则指向该块的结束点）的控制转移上， v 的明确赋值状态与块语句、`checked` 语句或 `unchecked` 语句之前的 v 的明确赋值状态相同。

5.3.3.3 表达式语句

对于由表达式 `expr` 组成的表达式语句 `stmt`：

- v 在 `expr` 的开头处与在 `stmt` 的开头处具有相同的明确赋值状态。
- 如果 v 在 `expr` 的结尾处明确赋值，则它在 `stmt` 的结束点也明确赋值；否则，它在 `stmt` 的结束点也不明确赋值。

5.3.3.4 声明语句

- 如果 `stmt` 是不带有初始值设定项的声明语句，则 v 在 `stmt` 的结束点与在 `stmt`

的开头处具有相同的明确赋值状态。

- 如果 `stmt` 是带有初始值设定项的声明语句，则确定 v 的明确赋值状态时可把 `stmt` 当做一个语句列表，其中每个带有初始值设定项的声明对应一个赋值语句（按声明的顺序）。

5.3.3.5 if 语句

对于具有以下形式的 `if` 语句 `stmt`：

`if (expr) then-stmt else else-stmt`

- v 在 `expr` 的开头处与在 `stmt` 的开头处具有相同的明确赋值状态。
- 如果 v 在 `expr` 的结尾处明确赋值，则它在指向 `then-stmt` 和 `else-stmt` 或指向 `stmt` 的结束点（如果没有 `else` 子句）的控制流转移上是明确赋值的。
- 如果 v 在 `expr` 的结尾处具有“在 `true` 表达式后明确赋值”状态，则它在指向 `then-stmt` 的控制流转移上是明确赋值的，在指向 `else-stmt` 或指向 `stmt` 的结束点（如果没有 `else` 子句）的控制流转移上不是明确赋值的。
- 如果 v 在 `expr` 的结尾处具有“在 `false` 表达式后明确赋值”状态，则它在指向 `else-stmt` 的控制流转移上是明确赋值的，在指向 `then-stmt` 的控制流转移上不是明确赋值的。此后，当且仅当它在 `then-stmt` 的结束点是明确赋值的时，它在 `stmt` 的结束点才是明确赋值的。
- 否则，认为 v 在指向 `then-stmt` 或 `else-stmt`，或指向 `stmt` 的结束点（如果没有 `else` 子句）的控制流转移上都不是明确赋值的。

5.3.3.6 switch 语句

在带有控制表达式 `expr` 的 `switch` 语句 `stmt` 中：

- 位于 `expr` 开头处的 v 的明确赋值状态与位于 `stmt` 开头处的 v 的明确赋值状态相同。
- 在指向可访问的 `switch block` 语句列表的控制流转移上， v 的明确赋值状态就是它在 `expr` 结尾处的明确赋值状态。

5.3.3.7 while 语句

对于具有以下形式的 `while` 语句 `stmt`：

`while (expr) while-body`

- v 在 `expr` 的开头处与在 `stmt` 的开头处具有相同的明确赋值状态。
- 如果 v 在 `expr` 的结尾处明确赋值，则它在指向 `while-body` 和指向 `stmt` 结束点的控制流转移上是明确赋值的。
- 如果 v 在 `expr` 的结尾处具有“在 `true` 表达式后明确赋值”状态，则它在指向 `while-body` 的控制流转移上是明确赋值的，但在 `stmt` 的结束点处不是明确赋值的。
- 如果 v 在 `expr` 的结尾处具有“在 `false` 表达式后明确赋值”状态，则它在指向 `stmt` 的结束点的控制流转移上是明确赋值的，但在指向 `while-body` 的控制流转移上不是明确赋值的。

5.3.3.8 do 语句

对于具有以下形式的 do 语句 stmt:

```
do do-body while ( expr );
```

- v 在从 stmt 的开头处到 do-body 的控制流转移上的明确赋值状态与在 stmt 的开头处的状态相同。
- v 在 expr 的开头处与在 do-body 的结束点具有相同的明确赋值状态。
- 如果 v 在 expr 的结尾处是明确赋值的, 则它在指向 stmt 的结束点的控制流转移上是明确赋值的。
- 如果 v 在 expr 的结尾处的状态为“在 false 表达式后明确赋值”, 则它在指向 stmt 的结束点的控制流转移上是明确赋值的。

5.3.3.9 for 语句

对具有以下形式的 for 语句进行的明确赋值检查:

```
for ( for-initializer ; for-condition ; for-iterator ) embedded-statement
```

就如执行下列语句一样:

```
{
    for-initializer ;
    while ( for-condition ) {
        embedded-statement ;
        for-iterator ;
    }
}
```

如果 for 语句中省略了 for-condition, 则在确定关于明确赋值的状态时, 可把上述展开语句列表中的 for-condition 当做 true。

5.3.3.10 break 语句、continue 语句和 goto 语句

由 break 语句、continue 语句或 goto 语句引起的控制流转移上的 v 的明确赋值状态与它在该语句开头处的明确赋值状态是一样的。

5.3.3.11 throw 语句

对于具有以下形式的语句 stmt

```
throw expr ;
```

位于 expr 开头处的 v 的明确赋值状态与位于 stmt 开头处的 v 的明确赋值状态相同。

5.3.3.12 return 语句

对于具有以下形式的语句 stmt

```
return expr ;
```

- 位于 expr 开头处的 v 的明确赋值状态与位于 stmt 开头处的 v 的明确赋值状态相同。
- 如果 v 是输出参数, 则它必须在下列两个位置之一被明确赋值。

- 在 `expr` 之后。
- 在包含 `return` 语句的 `try-finally` 或 `try-catch-finally` 的 `finally` 块的结尾处。

5.3.3.13 try-catch 语句

对于具有以下形式的语句 `stmt`:

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
```

- 位于 `try` 块开头处的 v 的明确赋值状态与位于 `stmt` 开头处的 v 的明确赋值状态相同。
- 位于 `catch-block-i` (对于所有的 i) 开头处的 v 的明确赋值状态与位于 `stmt` 开头处的 v 的明确赋值状态相同。
- 当且仅当 v 在 `try-block` 和每个 `catch-block-i` (每个 i 从 1 到 n) 的结束点明确赋值时, `stmt` 结束点处的 v 的明确赋值状态才是明确赋值的。

5.3.3.14 try-finally 语句

对于具有以下形式的 `try` 语句 `stmt`:

```
try try-block finally finally-block
```

- 位于 `try` 块开头处的 v 的明确赋值状态与位于 `stmt` 开头处的 v 的明确赋值状态相同。
- 位于 `finally` 块开头处的 v 的明确赋值状态与位于 `stmt` 开头处的 v 的明确赋值状态相同。
- 当且仅当下列条件中至少有一个为真时, 位于 `stmt` 结束点的 v 的明确赋值状态才是明确赋值的。
 - v 在 `try` 块的结束点明确赋值。
 - v 在 `finally` 块的结束点明确赋值。

如果控制流转移(例如 `goto` 语句)从 `try` 块内开始, 在 `try` 块外结束, 那么如果 v 在 `finally` 块的结束点明确赋值, 则 v 也被认为在该控制流转移上明确赋值(这不是必要条件, 如果 v 由于其他原因在该控制流转移上明确赋值, 则它仍被视为明确赋值)。

5.3.3.15 try-catch-finally 语句

对于具有以下形式的 `try-catch-finally` 语句:

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
finally finally-block
```

在进行明确赋值分析时, 可把该语句当做包含了 `try-catch` 语句的 `try-finally` 语句, 如下所示:

```

    try {
    try try-block
    catch(...) catch-block-1
    ...
    catch(...) catch-block-n
}
finally finally-block

```

下面的示例说明 `try` 语句 (§ 8.1) 的不同块如何影响明确赋值状态。

```

class A
{
    static void F() {
        int i, j;
        try {
            goto LABEL;
            // i 和 j 都不是明确赋值的
            i = 1;
            // i 是明确赋值的
        }
        catch {
            // i 和 j 都不是明确赋值的
            i = 3;
            // i 是明确赋值的
        }
        finally {
            // i 和 j 都不是明确赋值的
            j = 5;
            // j 是明确赋值的
        }
        // i 和 j 是明确赋值的
    LABEL:;
        // j 是明确赋值的
    }
}

```

5.3.3.16 foreach 语句

对于具有以下形式的 `foreach` 语句 `stmt`:

`foreach (type identifier in expr) embedded-statement`

- 位于 `expr` 开头处的 `v` 的明确赋值状态与位于 `stmt` 开头处的 `v` 的明确赋值状态相同。
- 在指向 `embedded-statement` 或指向 `stmt` 结束点处的控制流转移上, `v` 的明确赋值状态与位于 `expr` 结尾处的 `v` 的状态相同。

5.3.3.17 using 语句

对于具有以下形式的 `using` 语句 `stmt`:

`using (resource-acquisition) embedded-statement`

- 位于 `resource-acquisition` 开头处的 `v` 的明确赋值状态与位于 `stmt` 开头处的 `v` 的明确赋值状态相同。
- 在指向 `embedded-statement` 的控制流转移上, `v` 的明确赋值状态与位于

resource-acquisition 结尾处的 v 的状态相同。

5.3.3.18 lock 语句

对于具有以下形式的 lock 语句 stmt:

lock (expr) embedded-statement

- 位于 expr 开头处的 v 的明确赋值状态与位于 stmt 开头处的 v 的明确赋值状态相同。
- 在指向 embedded-statement 的控制流转移上, v 的明确赋值状态与位于 expr 结尾处的 v 的状态相同。

5.3.3.19 简单表达式的一般规则

下列规则应用于这些表达式: 文本 (§ 7.5.1)、简单名称 (§ 7.5.2)、成员访问表达式 (§ 7.5.4)、非索引 base 访问表达式 (§ 7.5.8) 和 typeof 表达式 (§ 7.5.11)。

- 位于此类表达式结尾处的 v 的明确赋值状态与位于表达式开头处的 v 的明确赋值状态相同。

5.3.3.20 带有嵌入表达式的表达式的一般规则

下列规则应用于这些表达式: 带括号的表达式 (§ 7.5.3)、元素访问表达式 (§ 7.5.6)、带有索引的 base 访问表达式 (§ 7.5.8)、增量和减量表达式 (§ 7.5.9, § 7.6.5)、强制转换表达式 (§ 7.6.6)、一元 +, -, ~, * 表达式, 二元 +, -, *, /, %, <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ 表达式 (§ 7.7, § 7.8, § 7.9, § 7.10)、复合赋值表达式 (§ 7.13.2)、checked 和 unchecked 表达式 (§ 7.5.12)、数组和委托创建表达式 (§ 7.5.10)。

这些表达式的每一个都有一个和多个按固定顺序无条件计算的子表达式。例如, 二元运算符 % 先计算运算符左边的值, 然后计算右边的值。索引操作先计算索引表达式, 然后按从左到右的顺序计算每个索引表达式。对于具有子表达式 $expr_1, expr_2, \dots, expr_n$ 的表达式 expr, 按下列顺序计算:

- 位于 $expr_1$ 开头处的 v 的明确赋值状态与位于 expr 开头处的 v 的明确赋值状态相同。
- 位于 $expr_i$ (i 大于 1) 开头处的 v 的明确赋值状态与位于 $expr_{i-1}$ 结尾处的 v 的明确赋值状态相同。
- 位于 expr 结尾处的 v 的明确赋值状态与位于 $expr_n$ 结尾处的 v 的明确赋值状态相同。

5.3.3.21 调用表达式和对象创建表达式

对于具有以下形式的调用表达式 expr:

primary-expression (arg₁, arg₂, ..., arg_n)

或具有以下形式的对象创建表达式:

new type (arg₁, arg₂, ..., arg_n)

- 对于调用表达式, 位于 primary-expression 之前的 v 的明确赋值状态与位于 expr 之前的 v 的状态相同。

- 对于调用表达式, 位于 arg_i 之前的 v 的明确赋值状态与位于 $\text{primary-expression}$ 之后的 v 的明确赋值状态相同。
- 对于对象创建表达式, 位于 arg_i 之前的 v 的明确赋值状态与位于 expr 之前的 v 的状态相同。
- 对于每一个参数 arg_i , 位于 arg_i 之后的 v 的明确赋值状态由正则表达式规则决定, 其中忽略所有的 ref 或 out 修饰符。
- 对于每一个 i 大于 1 的参数 arg_i , 位于 arg_i 之前的 v 的明确赋值状态与位于 arg_{i-1} 之后的 v 的状态相同。
- 如果变量 v 是被作为 out 参数传送 (形式为 “ $\text{out } v$ ” 的参数) 的, 则无论将它用做哪一个 arg_i , 在 expr 之后, v 的状态都是明确赋值的。否则, 位于 expr 之后的 v 的状态与位于 arg_n 之后的 v 的状态相同。

5.3.3.22 简单赋值表达式

对于具有形式 $w = \text{expr-rhs}$ 的表达式 expr :

- 位于 expr-rhs 之前的 v 的明确赋值状态与位于 expr 之前的 v 的明确赋值状态相同。
- 如果 w 与 v 是同一变量, 则位于 expr 之后的 v 的明确赋值状态是明确赋值的。否则, 位于 expr 之后的 v 的明确赋值状态与位于 expr-rhs 之后的 v 的明确赋值状态相同。

5.3.3.23 && 表达式

对于形式为 $\text{expr-first} \ \&\& \ \text{expr-second}$ 的表达式 expr :

- 位于 expr-first 之前的 v 的明确赋值状态与位于 expr 之前的 v 的明确赋值状态相同。
- 如果位于 expr-first 之后的 v 的状态是明确赋值的或者为 “在 true 表达式后明确赋值”, 则位于 expr-second 之前的 v 的明确赋值状态是明确赋值的。否则, 它就不是明确赋值的。
- 位于 expr 之后的 v 的明确赋值状态由以下内容决定。
 - 如果在 expr-first 之后的 v 的状态是明确赋值的, 则在 expr 之后的 v 的状态也是明确赋值的。
 - 否则, 如果位于 expr-second 之后的 v 的状态是明确赋值的, 而且位于 expr-first 之后的 v 的状态为 “在 false 表达式后明确赋值”, 则位于 expr 之后的 v 的状态是明确赋值的。
 - 否则, 如果位于 expr-second 之后的 v 的状态是明确赋值的或为 “在 true 表达式后明确赋值”, 则位于 expr 之后的 v 的状态是 “在 true 表达式后明确赋值”。
 - 否则, 如果位于 expr-first 之后的 v 的状态是 “在 false 表达式后明确赋值”, 而且位于 expr-second 之后的 v 的状态是 “在 false 表达式后明确赋值”, 则位于 expr 之后的 v 的状态是 “在 false 表达式后明确赋值”。

- 否则，在 `expr` 之后的 `v` 的状态不是明确赋值的。

下面是一个示例：

```
class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i 是明确赋值的
        }
        else {
            // i 不是明确赋值的
        }
        // i 不是明确赋值的
    }
}
```

其中，变量 `i` 被视为在 `if` 语句的一个嵌入语句中已明确赋值，而在另一个嵌入语句中未明确赋值。在 `F` 方法中的 `if` 语句中，由于总是在第一个嵌入语句执行前执行表达式 `(i = y)`，因此变量 `i` 在第一个嵌入语句中已明确赋值。相反，变量 `i` 在第二个嵌入语句中没有明确赋值，因为 `x >= 0` 可能已测试为 `false`，从而导致变量 `i` 未赋值。

5.3.3.24 || 表达式

对于形式为 `expr-first || expr-second` 的表达式 `expr`：

- 位于 `expr-first` 之前的 `v` 的明确赋值状态与位于 `expr` 之前的 `v` 的明确赋值状态相同。
- 如果位于 `expr-first` 之后的 `v` 的状态是明确赋值的或“在 `false` 表达式后明确赋值”，则位于 `expr-second` 之前的 `v` 的明确赋值状态是明确赋值的。否则，它就不是明确赋值的。
- 位于 `expr` 之后的 `v` 的明确赋值状态取决于：
 - 如果在 `expr-first` 之后，`v` 的状态是明确赋值的，则在 `expr` 之后的 `v` 的状态也是明确赋值的。
 - 否则，如果位于 `expr-second` 之后的 `v` 的状态是明确赋值的，而且位于 `expr-first` 之后的 `v` 的状态为“在 `true` 表达式后明确赋值”，则位于 `expr` 之后的 `v` 的状态是明确赋值的。
 - 否则，如果位于 `expr-second` 之后的 `v` 的状态是明确赋值的或是“在 `false` 表达式后明确赋值”，则位于 `expr` 之后的 `v` 的状态是“在 `false` 表达式后明确赋值”。
 - 否则，如果位于 `expr-first` 之后的 `v` 的状态是“在 `true` 表达式后明确赋值”，而且位于 `expr-second` 之后的 `v` 的状态是“在 `true` 表达式后明确赋值”，则位于 `expr` 之后的 `v` 的状态是“在 `true` 表达式后明确赋值”。
 - 否则，在 `expr` 之后，`v` 的状态就不是明确赋值的。

下面是一个示例：

```
class A
{
```



```

static void G(int x, int y) {
    int i;
    if (x >= 0 || (i = y) >= 0) {
        // i 不是明确赋值的
    }
    else {
        // i 是明确赋值的
    }
    // i 不是明确赋值的
}
}

```

变量 *i* 被视为在 `if` 语句的一个嵌入语句中已明确赋值，而在另一个嵌入语句中未明确赋值。在 `G` 方法中的 `if` 语句中，由于总是在第二个嵌入语句执行前执行表达式 `(i = y)`，因此变量 *i* 在第二个嵌入语句中已明确赋值。相反，在第一个嵌入语句中，变量 *i* 的状态不是明确赋值的，因为 `x >= 0` 可能已测试为 `true`，从而导致变量 *i* 未赋值。

5.3.3.25 ! 表达式

对于形式为 `!expr-operand` 的表达式 `expr`:

- 位于 `expr-operand` 之前的 *v* 的明确赋值状态与位于 `expr` 之前的 *v* 的明确赋值状态相同。
- 位于 `expr` 之后的 *v* 的明确赋值状态取决于以下内容。
 - 如果在 `expr-operand` 之后，*v* 的状态是明确赋值的，则在 `expr` 之后，*v* 的状态也是明确赋值的。
 - 如果在 `expr-operand` 之后，*v* 的状态不是明确赋值的，则在 `expr` 之后，*v* 的状态也不是明确赋值的。
 - 如果位于 `expr-operand` 之后的 *v* 的状态是“在 `false` 表达式后明确赋值”，则位于 `expr` 之后的 *v* 的状态是“在 `true` 表达式后明确赋值”。
 - 如果位于 `expr-operand` 之后的 *v* 的状态是“在 `true` 表达式后明确赋值”，则位于 `expr` 之后的 *v* 的状态是“在 `false` 表达式后明确赋值”。

5.3.3.26 ?: 表达式

对于形式为 `expr-cond ? expr-true : expr-false` 的表达式 `expr`:

- 位于 `expr-cond` 之前的 *v* 的明确赋值状态与位于 `expr` 之前的 *v* 的状态相同。
- 当且仅当位于 `expr-cond` 之后的 *v* 的状态是明确赋值的或“在 `true` 表达式后明确赋值”时，位于 `expr-true` 之前的 *v* 的明确赋值状态才是明确赋值的。
- 当且仅当位于 `expr-cond` 之后的 *v* 的状态是明确赋值的或“在 `false` 表达式后明确赋值”时，位于 `expr-false` 之前的 *v* 的明确赋值状态才是明确赋值的。
- 位于 `expr` 之后的 *v* 的明确赋值状态取决于以下内容。
 - 如果 `expr-cond` 是值为 `true` 的常数表达式 (§ 7.15)，则位于 `expr` 之后的 *v* 的状态与位于 `expr-true` 之后的 *v* 的状态相同。
 - 否则，如果 `expr-cond` 是值为 `false` 的常数表达式 (§ 7.15)，则位于 `expr` 之后的 *v* 的状态与位于 `expr-false` 之后的 *v* 的状态相同。
 - 否则，如果位于 `expr-true` 之后的 *v* 的状态是明确赋值的，而且位于 `expr-false`

之后的 v 的状态也是明确赋值的，则位于 `expr` 之后的 v 的状态是明确赋值的。

- 否则，在 `expr` 之后， v 的状态不是明确赋值的。

5.4 变量引用

变量引用是一个表达式，它被归类为变量。变量引用表示一个存储位置，访问它可以获取当前值及存储新值。

variable-reference: (变量引用:)

expression (表达式)

在 C 和 C++ 中，变量引用称为 `lvalue`。

5.5 变量引用的原子性

下列数据类型的读写是原子的：`bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `uint`, `int`, `float` 和引用类型。除此之外，当枚举类型的基础类型属于上述类型之一时，对它的读写也是原子的。其他类型的读写，包括 `long`, `ulong`, `double` 和 `decimal`，以及用户定义类型，都不保证是原子的。除专为该目的设计的库函数以外，对于增量或减量这类操作也不能保证进行原子的读、修改和写。

第 6 章 转换

转换 (conversion) 使一种类型的表达式可以被视为另一种类型。转换可以是**隐式的 (implicit)** 或**显式的 (explicit)**，这将确定是否需要显式的强制转换。例如，从 `int` 类型到 `long` 类型的转换是隐式的，因此 `int` 类型的表达式可隐式地按 `long` 类型处理。从 `long` 类型到 `int` 类型的反向转换是显式的，因此需要显式的强制转换。

```
int a = 123;
long b = a;      //从 int 到 long 的隐式转换
int c = (int) b; //从 long 到 int 的显式转换
```

某些转换由语言定义。程序也可以定义自己的转换 (§ 6.4)。

6.1 隐式转换

下列转换属于隐式转换：

- 标识转换
- 隐式数值转换
- 隐式枚举转换
- 隐式引用转换
- 装箱转换
- 隐式常数表达式转换
- 用户定义的隐式转换

隐式转换可以在各种情况下发生，包括函数成员调用 (§ 7.4.3)、强制转换表达式 (§ 7.6.6) 和赋值 (§ 7.13)。

预定义的隐式转换总是会成功，从来不会导致引发异常。正确设计的用户定义的隐式转换同样应表现出这些特性。

6.1.1 标识转换

标识转换是在同一类型（可为任何类型）内进行转换。这种转换的存在，仅仅是为了使已具有所需类型的实体可被认为是可转换的（转换为该类型）。

6.1.2 隐式数值转换

隐式数值转换为：

- 从 sbyte 到 short, int, long, float, double 或 decimal。
- 从 byte 到 short, ushort, int, uint, long, ulong, float, double 或 decimal。
- 从 short 到 int, long, float, double 或 decimal。
- 从 ushort 到 int, uint, long, ulong, float, double 或 decimal。
- 从 int 到 long, float, double 或 decimal。
- 从 uint 到 long, ulong, float, double 或 decimal。
- 从 long 到 float, double 或 decimal。
- 从 ulong 到 float, double 或 decimal。
- 从 char 到 ushort, int, uint, long, ulong, float, double 或 decimal。
- 从 float 到 double。

从 int, uint, long 或 ulong 到 float, 以及从 long 或 ulong 到 double 的转换可能导致精度损失, 但绝不会影响到它的数量级。其他的隐式数值转换绝不会丢失任何信息。

不存在向 char 类型的隐式转换, 因此其他整型的值不会自动转换为 char 类型。

6.1.3 隐式枚举转换

隐式枚举转换允许将十进制整数 0 转换为任何枚举类型。

6.1.4 隐式引用转换

隐式引用转换为:

- 从任何引用类型到 object。
- 从任何类类型 S 到任何类类型 T (前提是 S 是从 T 派生的)。
- 从任何类类型 S 到任何接口类型 T (前提是 S 实现了 T)。
- 从任何接口类型 S 到任何接口类型 T (前提是 S 是从 T 派生的)。
- 从元素类型为 S_E 的数组类型 S 到元素类型为 T_E 的数组类型 T (前提是以下所列的条件均为真)。
 - S 和 T 只是元素类型不同。换言之, S 和 T 具有相同的维数。
 - S_E 和 T_E 都是引用类型。
 - 存在从 S_E 到 T_E 的隐式引用转换。
- 从任何数组类型到 System.Array。
- 从任何委托类型到 System.Delegate。
- 从 null 类型到任何引用类型。

隐式引用转换是指一类引用类型之间的转换, 这种转换总是可以成功, 因此不需要在运行时进行任何检查。

引用转换无论是隐式的还是显式的, 都不会更改所转换的对象的引用标识。换言之, 虽然引用转换可能改变该引用的类型, 但绝不会更改所引用对象的类型或值。

6.1.5 装箱转换

装箱转换允许将值类型隐式转换为引用类型。将值类型的一个值装箱包括以下操作：分配一个对象实例，然后将值类型的值复制到该实例中。

有关装箱转换的介绍详见 § 4.3.1。

6.1.6 隐式常数表达式转换

隐式常数表达式转换允许进行以下转换：

- `int` 类型的常数表达式（§ 7.15）可以转换为 `sbyte`, `byte`, `short`, `ushort`, `uint` 或 `ulong` 类型（前提是常数表达式的值在目标类型的范围内）。
- `long` 类型的常数表达式可以转换为 `ulong` 类型（前提是常数表达式的值不为负）。

6.1.7 用户定义的隐式转换

用户定义的隐式转换由以下三部分组成：先是一个标准的隐式转换（可选）；然后是执行用户定义的隐式转换运算符；最后是另一个标准的隐式转换（可选）。计算用户定义的转换的精确规则详见 § 6.4.3 中的说明。

6.2 显式转换

下列转换属于显式转换：

- 所有隐式转换
- 显式数值转换
- 显式枚举转换
- 显式引用转换
- 显式接口转换
- 取消装箱转换
- 用户定义的显式转换

显式转换可在强制转换表达式（§ 7.6.6）中出现。

显式转换集包括所有隐式转换。这意味着允许使用冗余的强制转换表达式。

不是隐式转换的显式转换是这样的一类转换：它们不能保证总是成功，知道有可能丢失信息，变换前后的类型显著不同，从而值得使用显式表示法。

6.2.1 显式数值转换

显式数值转换是指从一个数值类型到另一个数值类型的转换，此转换不能用已知的隐式数值转换 (§ 6.1.2) 实现，它包括：

- 从 sbyte 到 byte, ushort, uint, ulong 或 char。
- 从 byte 到 sbyte 和 char。
- 从 short 到 sbyte, byte, ushort, uint, ulong 或 char。
- 从 ushort 到 sbyte, byte, short 或 char。
- 从 int 到 sbyte, byte, short, ushort, uint, ulong 或 char。
- 从 uint 到 sbyte, byte, short, ushort, int 或 char。
- 从 long 到 sbyte, byte, short, ushort, int, uint, ulong 或 char。
- 从 ulong 到 sbyte, byte, short, ushort, int, uint, long 或 char。
- 从 char 到 sbyte, byte 或 short。
- 从 float 到 sbyte, byte, short, ushort, int, uint, long, ulong, char 或 decimal。
- 从 double 到 sbyte, byte, short, ushort, int, uint, long, ulong, char, float 或 decimal。
- 从 decimal 到 sbyte, byte, short, ushort, int, uint, long, ulong, char, float 或 double。

由于显式转换包括所有隐式和显式数值转换，因此总是可以使用强制转换表达式 (§ 7.6.6) 从任何数值类型转换为任何其他数值类型。

显式数值转换有可能丢失信息或导致引发异常。显式数值转换按下面所述处理。

- 对于从一个整型到另一个整型的转换，处理取决于该转换发生时的溢出检查上下文 (§ 7.5.12)。
 - ◆ 在 checked 上下文中，如果源操作数的值在目标类型的范围内，转换就会成功。但如果源操作数的值在目标类型的范围外，则会引发 `System.OverflowException`。
 - ◆ 在 unchecked 上下文中，转换总是会成功并按下面所述进行。
 - 如果源类型大于目标类型，则截断源值（截去源值中容不下的最高有效位）。然后将结果视为目标类型的值。
 - 如果源类型小于目标类型，则源值或按符号扩展或按零扩展，以使它的大小与目标类型相同。如果源类型是有符号的，则使用按符号扩展；如果源类型是无符号的，则使用按零扩展。然后将结果视为目标类型的值。
 - 如果源类型的大小与目标类型相同，则源值被视为目标类型的值。
- 对于从 decimal 到整型的转换，源值向零舍入到最接近的整数值，该整数值成为转换的结果。如果转换得到的整数值不在目标类型的范围内，则会引发 `System.OverflowException`。
- 对于从 float 或 double 到整型的转换，处理取决于发生该转换时的溢出检查上下文 (§ 7.5.12)。

- ◆ 在 `checked` 上下文中，按如下所示进行转换。
 - 如果操作数的值是 NaN 或无穷大，则引发 `System.OverflowException`。
 - 否则，源操作数会向零舍入到最接近的整数值。如果该整数值处于目标类型的范围内，则该值就是转换的结果。
 - 否则，引发 `System.OverflowException`。
- ◆ 在 `unchecked` 上下文中，转换总是会成功并按下面这样继续。
 - 如果操作数的值是 NaN 或 `infinite`，则转换的结果是目标类型的一个未经指定的值。
 - 否则，源操作数会向零舍入到最接近的整数值。如果该整数值处于目标类型的范围内，则该值就是转换的结果。
 - 否则，转换的结果是目标类型的一个未经指定的值。
- 对于从 `double` 到 `float` 的转换，`double` 值舍入到最接近的 `float` 值。如果 `double` 值过小，无法表示为 `float` 值，则结果变成正零或负零。如果 `double` 值过大，无法表示为 `float` 值，则结果变成正无穷大或负无穷大。如果 `double` 值为 NaN，则结果仍然是 NaN。
- 对于从 `float` 或 `double` 到 `decimal` 的转换，源值转换为用 `decimal` 形式来表示，并且在需要时，将它在第 28 位小数位上舍入到最接近的数字（§ 4.1.7）。如果源值过小，无法表示为 `decimal`，则结果变成零。如果源值为 NaN、无穷大或者太大而无法表示为 `decimal` 值，则将引发 `System.OverflowException`。
- 对于从 `decimal` 到 `float` 或 `double` 的转换，`decimal` 值舍入到最接近的 `double` 或 `float` 值。虽然这种转换可能会损失精度，但绝不会导致引发异常。

6.2.2 显式枚举转换

显式枚举转换为：

- 从 `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double` 或 `decimal` 到任何枚举类型。
- 从任何枚举类型到 `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double` 或 `decimal`。
- 从任何枚举类型到任何其他枚举类型。

两种类型之间的显式枚举转换是通过将任何参与的枚举类型都按该枚举类型的基础类型处理，然后在结果类型之间执行隐式或显式数值转换。例如，给定具有 `int` 基础类型的枚举类型 `E`，从 `E` 到 `byte` 的转换按从 `int` 到 `byte` 的显式数值转换（§ 6.2.1）处理，而从 `byte` 到 `E` 的转换按从 `byte` 到 `int` 的隐式数值转换（§ 6.1.2）处理。

6.2.3 显式引用转换

显式引用转换为：

- 从 `object` 到任何其他引用类型。

- 从任何类类型 S 到任何类类型 T (前提是 S 为 T 的基类)。
- 从任何类类型 S 到任何接口类型 T (前提是 S 未密封并且 S 不实现 T)。
- 从任何接口类型 S 到任何类类型 T (前提是 T 未密封或 T 实现 S)。
- 从任何接口类型 S 到任何接口类型 T (前提是 S 不是从 T 派生的)。
- 从元素类型为 S_E 的数组类型 S 到元素类型为 T_E 的数组类型 T (前提是以下所列条件均为真)。
 - S 和 T 只是元素类型不同。换言之, S 和 T 具有相同的维数。
 - S_E 和 T_E 都是引用类型。
 - 存在从 S_E 到 T_E 的显式引用转换。
- 从 `System.Array` 及它实现的接口到任何数组类型。
- 从 `System.Delegate` 及它实现的接口到任何委托类型。

显式引用转换是那些需要运行时检查以确保它们正确的引用类型之间的转换。

为了使显式引用转换在运行时成功, 源操作数的值必须为 `null`, 或源操作数所引用的对象的实际类型必须是一个可通过隐式引用转换 (§ 6.1.4) 转换为目标类型的类型。如果显式引用转换失败, 则将引发 `System.InvalidCastException`。

无论是隐式的还是显式的引用转换, 都不会更改被转换的对象的引用标识。换言之, 虽然引用转换可能更改引用的类型, 但绝不会更改所引用对象的类型或值。

6.2.4 取消装箱转换

取消装箱转换允许将引用类型显式转换为值类型。取消装箱操作包括以下两个步骤: 首先检查对象实例是否为给定值类型的一个装了箱的值, 然后将该值从实例中复制出来。

有关取消装箱转换的进一步介绍详见 § 4.3.2。

6.2.5 用户定义的显式转换

用户定义的显式转换由以下三个部分组成: 先是一个标准的显式转换 (可选), 然后是执行用户定义的隐式或显式转换运算符, 最后是另一个标准的显式转换 (可选)。计算用户定义的转换的确切规则详见 § 6.4.4 中的说明。

6.3 标准转换

标准转换是那些预先定义的转换, 它们可以作为用户定义转换的组成部分出现。

6.3.1 标准隐式转换

下列隐式转换属于标准隐式转换:

- 标识转换 (§ 6.1.1)

- 隐式数值转换 (§ 6.1.2)
- 隐式引用转换 (§ 6.1.4)
- 装箱转换 (§ 6.1.5)
- 隐式常数表达式转换 (§ 6.1.6)

标准隐式转换特别排除了用户定义的隐式转换。

6.3.2 标准显式转换

标准显式转换包括所有的标准隐式转换，以及一个显式转换的子集，该子集是由那些与已知的标准隐式转换反向的转换组成的。换言之，如果存在一个从 A 类型到 B 类型的标准隐式转换，则一定存在与其对应的两个标准显式转换（一个是从 A 类型到 B 类型，另一个是从 B 类型到 A 类型）。

6.4 用户定义的转换

C# 允许通过用户定义的转换来增加预定义的隐式和显式转换。用户定义的转换是通过在类类型和结构类型中声明转换运算符 (§ 10.9.3) 引入的。

6.4.1 允许的用户定义转换

C# 只允许声明某些用户定义的转换。具体说来，不可能重新定义已存在的隐式或显式转换。仅当以下条件皆为真时，才允许类或结构声明从源类型 S 到目标类型 T 的转换。

- S 和 T 是不同的类型。
- S 和 T 中有一个属于类类型或结构类型，且包含了运算符的声明。
- S 和 T 都不是 object 类型或接口类型。
- T 不是 S 的基类，S 也不是 T 的基类。

关于用户定义转换的限制在 § 10.9.3 中有进一步讨论。

6.4.2 用户定义的转换的计算

用户定义的转换将一个值从它所属的类型（称为**源类型[source type]**）转换为另一个类型（称为**目标类型[target type]**）。用户定义的转换的计算集中在查找符合特定的源类型和目标类型的最精确的用户定义转换运算符。此确定过程分为几个步骤。

- 查找考虑从中使用用户定义的转换运算符的类和结构集。此集由源类型及其基类和目标类型及其基类组成（隐式假定只有类和结构可以声明用户定义的运算符，并且不属于类的类型不具有任何基类）。
- 由该类型集确定适用的用户定义转换运算符。一个转换运算符如满足下述条件就是适用的：必须可以通过执行标准转换 (§ 6.3) 来使源类型转换为该运算符的操

作数所要求的类型，并且必须可以通过执行标准转换来使运算符的结果类型转换为目标类型。

- 由适用的用户定义运算符集，明确地确定哪一个运算符是最精确的。概括地讲，最精确的运算符是操作数类型“最接近”源类型并且结果类型“最接近”目标类型的运算符。后面的章节定义了建立最精确的用户定义转换运算符的确切规则。

确定了最精确的用户定义转换运算符后，用户定义转换的实际执行包括三个步骤：

- 首先，如果需要，执行一个标准转换，将源类型转换为用户定义转换运算符的操作数所要求的类型。
- 然后，调用用户定义转换运算符以执行转换。
- 最后，如果需要，再执行一个标准转换，将用户定义转换运算符的结果类型转换为目标类型。

用户定义转换的计算从不涉及一个以上的用户定义转换运算符。换言之，从 S 类型到 T 类型的转换绝不会首先执行从 S 到 X 的用户定义转换，然后执行从 X 到 T 的用户定义转换。

后面的章节给出了用户定义的隐式或显式转换的确切定义。这些定义使用下面的术语：

- 如果存在一个从 A 类型到 B 类型的标准隐式转换 (§ 6.3.1)，并且 A 和 B 都不是接口类型，则称 A 被 B 包含、称 B 包含 A。
- 在一个类型集中，包含程度最大的类型是指这样的类型，它包含了该类型集中的所有其他类型。如果没有一个类型包含所有其他类型，则类型集中没有包含程度最大的类型。用更直观的话讲，包含程度最大的类型是类型集中“最大”的类型，即可将每个其他类型都隐式转换为该类型的一个类型。
- 在一个类型集中，被包含程度最大的类型是指这样一个类型：它被该类型集中的所有其他类型所包含。如果没有一个类型被所有其他类型包含，则类型集中没有被包含程度最大的类型。用更直观的话讲，被包含程度最大的类型是类型集中“最小的”类型，即可隐式转换为每个其他类型的一个类型。

6.4.3 用户定义的隐式转换

从 S 类型到 T 类型的用户定义的隐式转换按下面这样处理。

- 查找类型集 D，从该类型集考虑用户定义的转换运算符。此集合由 S（如果 S 是类或结构）、S 的基类（如果 S 是类）和 T（如果 T 是类或结构）组成。
- 查找适用的用户定义转换运算符集合 U。集合 U 由用户定义的隐式转换运算符组成，这些运算符是在 D 中的类或结构内声明的，用于从包含 S 的类型转换为被 T 包含的类型。如果 U 为空，则转换未定义并且发生编译时错误。
- 在 U 中查找运算符的最精确的源类型 S_x ：
 - 如果 U 中的所有运算符都从 S 转换，则 S_x 为 S。
 - 否则， S_x 在 U 中运算符的合并目标类型集中是被包含程度最大的类型。如果找不到这样的被包含程度最大的类型，则转换是不明确的，并且发生编译时错误。

- 在 U 中查找运算符的最精确的目标类型 T_X :
 - 如果 U 中的所有运算符都转换为 T , 则 T_X 为 T 。
 - 否则, T_X 在 U 中运算符的合并目标类型集中是包含程度最大的类型。如果找不到这样的包含程度最大的类型, 则转换是不明确的, 并且发生编译时错误。
- 如果 U 中正好含有一个从 S_X 转换到 T_X 的用户定义转换运算符, 则这就是最精确的转换运算符。如果不存在此类运算符, 或者如果存在多个此类运算符, 则转换是不明确的, 并且发生编译时错误。否则, 将应用用户定义的转换。
 - 如果 S 不是 S_X , 则先执行一个从 S 到 S_X 的标准隐式转换。
 - 调用最精确的用户定义转换运算符, 以从 S_X 转换到 T_X 。
 - 如果 T_X 不是 T , 则再执行一个 T_X 到 T 的标准隐式转换。

6.4.4 用户定义的显式转换

从 S 类型到 T 类型的用户定义的显式转换按下面这样处理。

- 查找类型集 D , 从该类型集考虑用户定义的转换运算符。该类型集由 S (如果 S 为类或结构)、 S 的基类 (如果 S 为类)、 T (如果 T 为类或结构) 和 T 的基类 (如果 T 为类) 组成。
- 查找适用的用户定义转换运算符集合 U 。集合 U 由用户定义的隐式或显式转换运算符组成, 这些运算符是在 D 中的类或结构内声明的, 用于从包含 S 或被 S 包含的类型转换为包含 T 或被 T 包含的类型。如果 U 为空, 则转换未定义并且发生编译时错误。
- 在 U 中查找运算符的最精确的源类型 S_X :
 - 如果 U 中的所有运算符都从 S 转换, 则 S_X 为 S 。
 - 否则, 如果 U 中的所有运算符都从包含 S 的类型转换, 则 S_X 在这些运算符的合并源类型集中是被包含程度最大的类型。如果找不到最直接包含的类型, 则转换是不明确的, 并且发生编译时错误。
 - 否则, S_X 在 U 中运算符的合并源类型集中是包含程度最大的类型。如果找不到这样的包含程度最大的类型, 则转换是不明确的, 并且发生编译时错误。
- 在 U 中查找运算符的最精确的目标类型 T_X :
 - 如果 U 中的所有运算符都转换为 T , 则 T_X 为 T 。
 - 否则, 如果 U 中的所有运算符都转换为被 T 包含的类型, 则 T_X 在这些运算符的合并源类型集中是包含程度最大的类型。如果找不到这样的包含程度最大的类型, 则转换是不明确的, 并且发生编译时错误。
 - 否则, T_X 在 U 中运算符的合并目标类型集中是被包含程度最大的类型。如果找不到这样的被包含程度最大的类型, 则转换是不明确的, 并且发生编译时错误。
- 如果 U 中正好含有一个从 S_X 转换到 T_X 的用户定义转换运算符, 则这就是最精确的转换运算符。如果不存在此类运算符, 或者存在多个此类运算符, 则转换是不明确的, 并且发生编译时错误。否则, 将应用用户定义的转换:

- 如果 S 不是 S_X ，则先执行一个从 S 到 S_X 的标准显式转换。
- 调用最精确的用户定义转换运算符，以从 S_X 转换到 T_X 。
- 如果 T_X 不是 T ，则再执行一个从 T_X 到 T 的标准显式转换。

第 7 章 表达式

表达式是运算符和操作数的序列。本章将定义语法、操作数和运算符的计算顺序及表达式的含义。

7.1 表达式的分类

一个表达式可归类为下列类别之一：

- 值。每个值都有关联的类型。
- 变量。每个变量都有关联的类型，称为该变量的已声明类型。
- 命名空间。这类表达式只能出现在成员访问（§ 7.5.4）的左边。在任何其他上下文中，归类为命名空间的表达式将导致编译时错误。
- 类型。这类表达式只能出现在成员访问（§ 7.5.4）的左边，或作为 `as` 运算符（§ 7.9.10）、`is` 运算符（§ 7.9.9）或 `typeof` 运算符（§ 7.5.11）的操作数。在任何其他上下文中，归类为类型的表达式将导致编译时错误。
- 方法组，它是一组重载方法，是成员查找（§ 7.3）的结果。方法组可以有关联的实例表达式。当调用实例方法时，实例表达式的计算结果成为由 `this`（§ 7.5.7）表示的实例。只能在调用表达式（§ 7.5.5）或委托创建表达式（§ 7.5.10.3）中使用方法组。在任何其他上下文中，归类为方法组的表达式将导致编译时错误。
- 属性访问。每个属性访问都有关联的类型，即该属性的类型。此外，属性访问可以有关联的实例表达式。当调用实例属性访问的访问器（`get` 或 `set` 块）时，实例表达式的计算结果成为由 `this`（§ 7.5.7）表示的实例。
- 事件访问。每个事件访问都有关联的类型，即该事件的类型。此外，事件访问还可以有关联的实例表达式。事件访问可以作为 `+=` 和 `-=` 运算符的左操作数（§ 7.13.3）出现。在任何其他上下文中，归类为事件访问的表达式将导致编译时错误。
- 索引器访问。每个索引器访问都有关联的类型，即该索引器的元素类型。此外，索引器访问还可以有关联的实例表达式和关联的参数列表。当调用索引器访问的访问器（`get` 或 `set` 块）时，实例表达式的计算结果成为由 `this`（§ 7.5.7）表示的实例，而参数列表的计算结果成为调用的参数列表。
- `Nothing`。这出现在当表达式是调用一个具有 `void` 返回类型的方法时。`Nothing` 类别的表达式仅在语句表达式（§ 8.6）的上下文中有效。

表达式的最终结果绝不会是一个命名空间、类型、方法组或事件访问。这些类别的表达式是只能在特定上下文中使用的中间构造。

通过执行 `get` 访问器或 `set` 访问器的调用, 属性访问或索引器访问总是被重新归类为值。实际涉及哪个访问器由属性访问或索引器访问的上下文确定: 如果访问是赋值的目标, 则调用 `set` 访问器以赋新值 (§ 7.13.1)。否则调用 `get` 访问器以获取当前值 (§ 7.1)。

大多数含有表达式的构造最后都要求表达式表示一个值 (value)。在此情况下, 如果实际的表达式表示命名空间、类型、方法组或 `Nothing`, 则会发生编译时错误。但是, 如果表达式表示属性访问、索引器访问或变量, 则将它们隐式替换为相应的属性、索引器或变量的值。

- 变量的值只是当前存储在该变量所标识的存储位置的值。在可以获取变量的值之前, 变量必须被视为已明确赋值 (§ 5.3), 否则将发生编译时错误。
- 属性访问表达式的值通过调用属性的 `get` 访问器来获取。如果属性没有 `get` 访问器, 则发生编译时错误。否则将执行一个函数成员调用 (§ 7.4.3), 调用的结果成为属性访问表达式的值。
- 索引器访问表达式的值通过调用索引器的 `get` 访问器来获取。如果索引器没有 `get` 访问器, 则发生编译时错误。否则将使用与索引器访问表达式关联的参数列表来执行函数成员调用 (§ 7.4.3), 调用的结果成为索引器访问表达式的值。

7.2 运算符

表达式由操作数和运算符构成。表达式的运算符指示对操作数进行什么样的运算。运算符的示例包括 `+`, `-`, `*`, `/` 和 `new`。操作数的示例包括文本、字段、局部变量和表达式。有三类运算符:

- 一元运算符。一元运算符带一个操作数并使用前缀表示法 (如 `-x`) 或后缀表示法 (如 `x++`)。
- 二元运算符。二元运算符带两个操作数并且都使用中缀表示法 (如 `x + y`)。
- 三元运算符。只有一个三元运算符 `?:` 存在, 它带三个操作数并使用中缀表示法 (`c? x: y`)。

表达式中运算符的计算顺序由运算符的优先级和顺序关联性 (§ 7.2.1) 确定。

表达式中的操作数从左到右进行计算。例如, 在 `F(i) + G(i++) * H(i)` 中, 使用 `i` 的旧值调用 `F` 方法, 然后使用 `i` 的旧值调用 `G` 方法, 最后使用 `i` 的新值调用 `H` 方法。这与运算符的优先级无关。

某些运算符可以重载。运算符重载允许指定使用用户定义的运算符来执行某些运算, 这些运算的操作数中至少有一个属于用户定义类或结构类型 (§ 7.2.2)。

7.2.1 运算符的优先级和顺序关联性

当表达式包含多个运算符时, 运算符的优先级控制各运算符的计算顺序。例如, 表达式 `x + y * z` 按 `x + (y * z)` 计算, 因为运算符 `*` 具有的优先级比运算符 `+` 高。运算符的优先级由运算符的关联语法产生式的定义确定。例如, 一个增量表达式由以运算符 `+` 或 `-`

分隔的乘法表达式组成，因此给运算符+和 -赋予的优先级比运算符 *, / 和%低。

表 7.1 按照从高到低的优先级顺序概括了所有的运算符。

表 7.1 运算符的优先级顺序

章 节	类 别	运 算 符
7.5	基本	x.y f(x) a[x] x++ x--new typeof checked unchecked
7.6	一元	+ - !~ ++x --x (T)x
7.7	乘法	* / %
7.7	加法	+ -
7.8	移位	<< >>
7.9	关系和类型检测	< > <= >= is as
7.9	相等	== !=
7.10	逻辑与 AND	&
7.10	逻辑异或 XOR	^
7.10	逻辑或 OR	
7.11	条件与 AND	&&
7.11	条件或 OR	
7.12	条件	?:
7.13	赋值	= *= /= %= += -= <<= >>= &= ^= =

当操作数出现在具有相同优先级的两个运算符之间时，运算符的顺序关联性控制运算的执行顺序：

- 除了赋值运算符外，所有的二元运算符都向左顺序关联，意思是从左向右执行运算。例如，x + y + z 按 (x + y) + z 计算。
- 赋值运算符和条件运算符 (?:) 向右顺序关联，意思是从右向左执行运算。例如，x = y = z 按 x = (y = z) 计算。

优先级和顺序关联性都可以用括号控制。例如，x + y * z 先将 y 乘以 z 然后将结果与 x 相加，而 (x + y) * z 先将 x 与 y 相加，然后再将结果乘以 z。

7.2.2 运算符重载

所有一元和二元运算符都具有可自动用于任何表达式的预定义实现。除了预定义实现外，还可通过在类或结构中设置 operator 声明来引入用户定义的实现（§ 10.9）。用户定义的运算符实现的优先级总是高于预定义运算符实现的优先级：仅当没有适用的用户定义运算符实现时才会考虑预定义运算符实现。

可重载的一元运算符有：

+ - ! ~ ++ -- true false

虽然不在表达式中显式地使用 true 和 false，但仍将它们视为运算符，因为它们在某些表达式上下文中被调用：如布尔表达式（§ 7.16）和涉及条件运算符（§ 7.12）与条件逻辑运算符（§ 7.11）的表达式。

可重载的二元运算符有：

+ - * / % & | ^ << >> == != > < >= <=

只有以上所列的运算符可以重载。具体说来，不可能重载成员访问、方法调用或 `=`，`&&`，`||`，`?:`，`checked`，`unchecked`，`new`，`typeof`，`as` 和 `is` 运算符。

当重载一个二元运算符时，也会隐式重载相应的赋值运算符（若有）。例如，运算符 `*` 的重载也是运算符 `*=` 的重载。§ 7.13 对此有进一步描述。请注意，赋值运算符 `=` 本身不能重载。赋值总是简单地将值按位复制到变量中。

强制转换运算（如 `(T)x`）通过提供用户定义的转换（§ 6.4）来重载。

元素访问（如 `a[x]`）不被视为可重载的运算符。但是，可通过索引器（§ 10.8）支持用户定义的索引。

在表达式中，使用运算符表示法来引用运算符，而在声明中，使用函数表示法来引用运算符。表 7.2 显示了一元运算符和二元运算符的运算符表示法和函数表示法之间的关系。在第一项中，`op` 表示任何可重载的一元前缀运算符。在第二项中，`op` 表示 `++` 和 `--` 一元后缀运算符。在第三项中，`op` 表示任何可重载的二元运算符。

表 7.2 运算符表示法和函数表示法间的关系

运算符表示法	函数表示法
<code>op x</code>	<code>operator op (x)</code>
<code>x op</code>	<code>operator op(x)</code>
<code>x op y</code>	<code>operator op(x,y)</code>

用户定义的运算符声明总是要求至少一个参数为包含运算符声明的类类型或结构类型。因此，用户定义的运算符不可能具有与预定义运算符相同的签名。

用户定义的运算符声明不能修改运算符的语法、优先级或顺序关联性。例如，`/` 运算符总是一个二元运算符，总是具有在 § 7.2.1 中指定的优先级，并且总是向左顺序关联。

虽然用户定义的运算符可以执行它想执行的任何计算，但是强烈建议不要采用产生的结果与直觉预期不同的实现。例如，`operator ==` 的实现应比较两个操作数是否相等，然后返回一个适当的 `bool` 结果。

在从 § 7.5 到 § 7.13 的关于各运算符的说明中，对运算符的预定义实现及应用用于各运算符的任何其他规则都有规定。在这些说明中使用了“一元运算符重载决策”（`unary operator overload resolution`）、“二元运算符重载决策”（`binary operator overload resolution`）和“数值提升”（`numeric promotion`）这样的术语，在后面的章节中可以找到它们的定义。

7.2.3 一元运算符重载决策

`op x` 或 `x op` 形式的运算（其中 `op` 是可重载的一元运算符，`x` 是 `X` 类型的表达式）按下面这样处理：

- 对于由 `X` 为运算 `operator op(x)` 提供的候选的用户定义运算符的集合，应根据 §

7.2.5 中的规则来确定。

- 如果候选的用户定义运算符集合不为空，则它就会成为关于该运算的候选运算符集合。否则，预定义一元 operator op 实现成为关于该运算的候选运算符集合。关于给定运算符的预定义实现，在有关运算符的说明（§ 7.5 和 § 7.6）中指定。
- § 7.4.2 中的重载决策规则应用于候选运算符集合，以选择一个关于参数列表 (x) 的最好的运算符，此运算符将成为重载决策过程的结果。如果重载决策未能选出单个最佳运算符，则发生编译时错误。

7.2.4 二元运算符重载决策

x op y 形式的运算（其中 op 是可重载的二元运算符，x 是 X 类型的表达式，y 是 Y 类型的表达式）按下面这样处理：

- 确定 X 和 Y 为运算 operator op(x, y) 提供的候选用户定义运算符集合。该集合包括由 X 提供的候选运算符和由 Y 提供的候选运算符的联合，每个候选运算符由 § 7.2.5 中的规则确定。如果 X 和 Y 的类型相同，或者 X 和 Y 是从一个公共基类型派生的，则共享的候选运算符只在合并的集合中出现一次。
- 如果候选的用户定义运算符集合不为空，则它就会成为运算的候选运算符集合。否则，预定义二元 operator op 实现将成为该运算的候选运算符集合。关于给定运算符的预定义实现，在有关运算符的说明（§ 7.7 到 § 7.13）中指定。
- § 7.4.2 中的重载决策规则应用于候选运算符集合，以选择一个关于参数列表 (x, y) 的最好的运算符，此运算符将成为重载决策过程的结果。如果重载决策未能选出单个最佳运算符，则发生编译时错误。

7.2.5 候选用户定义运算符

给定一个 T 类型和运算 operator op(A)，其中 op 是可重载的运算符，A 是参数列表，对 T 为 operator op(A) 提供的候选用户定义运算符集合按下面这样确定：

- 对于 T 中的所有 operator op 声明，如果关于参数列表 A 至少有一个运算符是适用的（§ 7.4.2.1），则候选运算符集合由 T 中所有适用的 operator op 声明组成。
- 否则，如果 T 为 object，则候选运算符集合为空。
- 否则，由 T 提供的候选运算符集合是 T 的直接基类提供的候选运算符集合。

7.2.6 数值提升

数值提升包括自动为预定义一元和二元数值运算符的操作数执行某些隐式转换。数值提升不是一个独特的机制，而是一种将重载决策应用于预定义运算符所产生的效果。数值提升尤其不影响用户定义运算符的计算，尽管可以实现用户定义运算符以表现类似的效果。

作为数值提升的示例，请看二元运算符 * 的预定义实现：

```

int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);

```

当重载决策规则 (§ 7.4.2) 应用于此运算符集合时，这些运算符中第一个能满足下述条件的运算符会被选中：存在一个从给定的操作数的类型到它声明的类型的隐式转换。例如，对于 `b * s` 运算（其中 `b` 为 `byte`，`s` 为 `short`），重载决策选择 `operator *(int, int)` 作为最好的运算符。因此，效果是 `b` 和 `s` 转换为 `int`，结果的类型为 `int`。同样，对于 `i * d` 运算（其中 `i` 为 `int`，`d` 为 `double`），重载决策选择 `operator *(double, double)` 作为最好的运算符。

7.2.6.1 一元数值提升

一元数值提升是针对预定义的 `+`、`-` 和 `~` 一元运算符的操作数发生的。一元数值提升仅包括将 `sbyte`、`byte`、`short`、`ushort` 或 `char` 类型的操作数转换为 `int` 类型。此外，对于一元运算符 `-`，一元数值提升将 `uint` 类型的操作数转换为 `long` 类型。

7.2.6.2 二元数值提升

二元数值提升是针对预定义的 `+`、`-`、`*`、`/`、`%`、`&`、`|`、`^`、`==`、`!=`、`>`、`<`、`>=` 和 `<=` 二元运算符的操作数发生的。二元数值提升隐式地将两个操作数都转换为一个公共类型，如果涉及的是非关系运算符，则此公共类型还成为运算的结果类型。二元数值提升应按下列规则进行（以它们在此出现的顺序）：

- 如果有一个操作数的类型为 `decimal`，则另一个操作数转换为 `decimal` 类型；如果另一个操作数的类型为 `float` 或 `double`，则发生编译时错误。
- 如果有一个操作数的类型为 `double`，则另一个操作数转换为 `double` 类型。
- 如果有一个操作数的类型为 `float`，则另一个操作数转换为 `float` 类型。
- 如果有一个操作数的类型为 `ulong`，则另一个操作数转换为 `ulong` 类型；如果另一个操作数的类型为 `sbyte`、`short`、`int` 或 `long`，则发生编译时错误。
- 如果有一个操作数的类型为 `long`，则另一个操作数转换为 `long` 类型。
- 如果有一个操作数的类型为 `uint` 而另一个操作数的类型为 `sbyte`、`short` 或 `int`，则两个操作数都转换为 `long` 类型。
- 如果有一个操作数的类型为 `uint`，则另一个操作数转换为 `uint` 类型。
- 否则，两个操作数都转换为 `int` 类型。

请注意，第一个规则不允许将 `decimal` 类型与 `double` 类型和 `float` 类型混用。该规则遵循这样的事实：在 `decimal` 类型与 `double` 类型和 `float` 类型之间不存在隐式转换。

还需要注意的是，当一个操作数为有符号的整型时，另一个操作数的类型不可能为 `ulong`。原因是不存在一个既可以表示 `ulong` 的全部范围，又能表示有符号整数的整型类型。

在以上两种情况下，都可以使用强制转换表达式显式地将一个操作数转换为与另一个操作数兼容的类型。

在下面的示例中，

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

由于 `decimal` 类型不能与 `double` 类型相乘，因此发生编译时错误。通过将第二个操作数显式地转换为 `decimal` 消除此错误，如下所示：

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

7.3 成员查找

成员查找是确定类型上下文中的名称含义的过程。成员查找可能成为在表达式中计算简单名称（§ 7.5.2）或成员访问（§ 7.5.4）过程的组成部分。

`T` 类型中的名称 `N` 的成员查找按下面这样处理。

- 首先，构造一个在 `T` 和 `T` 的基类型（§ 7.3）中声明的名为 `N` 的所有可访问成员（§ 3.5）的集合。包含 `override` 修饰符的声明不包括在此集合中。如果名为 `N` 且可访问的成员不存在，则此查找不产生任何匹配，并且不对下面的步骤进行计算。
- 然后，从该集合中移除被其他成员隐藏的成员。对于该集合中的每个成员 `S.M`（其中 `S` 是声明了成员 `M` 的类型），应用下面的规则：
 - 如果 `M` 是常数、字段、属性、事件、类型或枚举成员，则从集合中移除在 `S` 的基类型中声明的所有成员。
 - 如果 `M` 是一个方法，则从集合中移除在 `S` 的基类型中声明的所有非方法成员，并从集合中移除与在 `S` 的基类型中声明的 `M` 具有相同签名的所有方法。
- 最后，移除了隐藏成员后，按下述规则确定查找的结果：
 - 如果集合由一个非方法成员组成，则此成员为查找的结果。
 - 如果集合只包含方法，则这组方法为查找的结果。
 - 否则，查找失败（无明确的结果），并发生编译时错误（只有对于具有多个直接基接口的接口中的成员查找才会出现这种情况）。

在不是接口的类型和严格单一继承的接口类型（继承链中的每个接口都只有零个或一个直接基接口）中，进行成员查找的规则可以简单地归结为：派生成员隐藏具有相同名称或签名的基成员。这种单一继承查找绝不会失败（一定有明确的结果）。有关多重继承中的成员查找可能引起的多义性的介绍详见 § 13.2.5。

出于成员查找的目的，类型 `T` 被视为具有下列基类型：

- 如果 `T` 为 `object`，则 `T` 没有基类型。
- 如果 `T` 为值类型，则 `T` 的基类型为类类型 `object`。
- 如果 `T` 为类类型，则 `T` 的基类型为 `T` 的基类，其中包括类类型 `object`。
- 如果 `T` 为接口类型，则 `T` 的基类型为 `T` 的基接口和类类型 `object`。

- 如果 T 为数组类型，则 T 的基类型为类类型 System.Array 和 object。
- 如果 T 为委托类型，则 T 的基类型为类类型 System.Delegate 和 object。

7.4 函数成员

函数成员是包含可执行语句的成员。函数成员总是类型的成员，不能是命名空间的成员。C# 定义了以下类别的函数成员：

- 方法
- 属性
- 事件
- 索引器
- 用户定义运算符
- 实例构造函数
- 静态构造函数
- 析构函数

除了析构函数和静态构造函数（它们不能被显式调用），函数成员中包含的语句通过函数成员调用执行。编写函数成员调用的实际语法取决于具体的函数成员类别。

函数成员调用中所带的参数列表（§ 7.4.1）提供了函数成员参数的实际值或变量引用。

调用方法、索引器、运算符和实例构造函数时，使用重载决策来确定要调用的候选函数成员集。有关此过程的介绍详见 § 7.4.2 。

在编译时（可能通过重载决策）确定了具体的函数成员后，有关运行时调用函数成员的实际过程的介绍详见 § 7.4.3 。

表 7.3 概述了在涉及 6 个可被显式调用的函数成员类别的构造中发生的处理过程。其中，e, x, y 和 value 代表变量或值类别的表达式，T 代表类型的表达式，F 是一个方法的简称，P 是一个属性的简称。

表 7.3 6 个构造中发生的处理过程

构造	示例	说明
方法调用	F(x, y)	应用重载决策以在包含类或结构中选择最佳的方法 F。用参数列表 (x, y) 调用该方法。如果该方法不为 static，则用 this 来表达对应的实例。
	T.F(x, y)	应用重载决策以在类或结构 T 中选择最佳的方法 F。如果该方法不为 static，则发生编译时错误。用参数列表 (x, y) 调用该方法。
	e.F(x, y)	应用重载决策以从 e 所属的类型确定的类、结构或接口中选择最佳的方法 F。如果该方法为 static，则发生编译时错误。用实例表达式 e 和参数列表 (x, y) 调用该方法。
属性访问	P	调用包含类或结构中属性 P 的 get 访问器。如果 P 是只写的，则发生编译时错误。如果 P 不为 static，则实例表达式为 this。
	P = value	用参数列表 (value) 调用包含类或结构中的属性 P 的 set 访问器。如果 P 是只读的，则发生编译时错误。如果 P 不为 static，则用 this 来表达对应的实例。

(续表)

构造	示例	说明
属性访问	T.P	调用类或结构 T 中属性 P 的 get 访问器。如果 P 不为 static，或者如果 P 是只写的，则发生编译时错误。
	T.P = value	用参数列表 (value) 调用类或结构 T 中的属性 P 的 set 访问器。如果 P 不为 static，或者如果 P 是只读的，则发生编译时错误。
	e.P	用实例表达式 e 调用由 e 的类型提供的类、结构或接口中属性 P 的 get 访问器。如果 P 为 static，或者如果 P 是只写的，则发生编译时错误。
	e.P = value	用实例表达式 e 和参数列表 (value) 调用 e 的类型给定的类、结构或接口中属性 P 的 set 访问器。如果 P 为 static，或者如果 P 是只读的，则发生编译时错误。
事件访问	E += value	调用包含类或结构中的事件 E 的 add 访问器。如果 E 不是静态的，则用 this 来表达对应的实例。
	E -= value	调用包含类或结构中事件 E 的 remove 访问器。如果 E 不是静态的，则用 this 来表达对应的实例。
	T.E += value	调用类或结构 T 中事件 E 的 add 访问器。如果 E 不是静态的，则发生编译时错误。
	T.E -= value	调用类或结构 T 中事件 E 的 remove 访问器。如果 E 不是静态的，则发生编译时错误。
	e.E += value	用实例表达式 e 调用由 e 的类型提供的类、结构或接口中事件 E 的 add 访问器。如果 E 是静态的，则发生编译时错误。
	e.E -= value	用实例表达式 e 调用由 e 的类型给定的类、结构或接口中事件 E 的 remove 访问器。如果 E 是静态的，则发生编译时错误。
索引器访问	e[x, y]	应用重载决策以在 e 的类型给定的类、结构或接口中选择最佳的索引器。用实例表达式 e 和参数列表 (x, y) 调用该索引器的 get 访问器。如果索引器是只写的，则发生编译时错误。
	e[x, y] = value	应用重载决策以在 e 的类型给定的类、结构或接口中选择最佳的索引器。用实例表达式 e 和参数列表 (x, y, value) 调用该索引器的 set 访问器。如果索引器是只读的，则发生编译时错误。
运算符调用	-x	应用重载决策以在 x 的类型给定的类或结构中选择最佳的一元运算符。用参数列表 (x) 调用选定的运算符。
	x + y	应用重载决策以在 x 和 y 的类型给定的类或结构中选择最佳的二元运算符。使用参数列表 (x, y) 调用选定的运算符。
实例构造函数	new T(x, y)	应用重载决策以在类或结构 T 中选择最佳的实例构造函数。用参数列表 (x, y) 调用该实例构造函数。

7.4.1 参数列表

每个函数成员调用均包括一个参数列表，该列表列出了函数成员参数的实际值或变量引用。如何指定函数成员调用的参数列表的语法取决于函数成员类别：

- 对于实例构造函数、方法和委托，参数被指定为参数列表。
- 对于属性，当调用 get 访问器时，参数列表是空的；而当调用 set 访问器时，参数列表由指定为赋值运算符的右操作数的表达式组成。
- 对于事件，参数列表由指定为 += 或 -= 运算符的右操作数的表达式组成。
- 对于索引器，参数列表由在索引器访问中的方括号之间指定的表达式组成。当调用 set 访问器时，参数列表还需附加上一个表达式，该表达式被指定为赋值运算

符的右操作数。

- 对于用户定义的运算符，参数列表由一元运算符的单个操作数或二元运算符的两个操作数组成。

对于属性 (§ 10.6)、事件 (§ 10.7)、索引器 (§ 10.8) 和用户定义的运算符 (§ 10.9) 来说，它们的参数总是作为值参数 (§ 10.5.1.1) 来传递。这些函数成员类别不支持引用参数和输出参数。

实例构造函数、方法或委托调用的参数按如下的参数列表形式指定：

argument-list: (参数列表:)

argument (参数)

argument-list , argument (参数列表 , 参数)

argument: (参数:)

expression (表达式)

ref variable-reference (ref 变量引用)

out variable-reference (out 变量引用)

参数列表由一个或多个参数组成，各参数之间用逗号隔开。每个参数都可以采用下列形式之一：

- 表达式，指示将该参数作为值参数 (§ 10.5.1.1) 传递。
- 后跟变量引用 (§ 5.4) 的关键字 `ref`，指示将参数作为引用参数 (§ 10.5.1.2) 传递。变量必须已明确赋值 (§ 5.3) 后才可作为引用参数传递。易失字段 (§ 10.4.3) 不能作为引用参数传递。
- 后跟变量引用 (§ 5.4) 的关键字 `out`，指示将参数作为输出参数 (§ 10.5.1.3) 传递。在将变量作为输出参数传递的函数成员调用之后，该变量被视为已明确赋值 (§ 5.3)。易失字段 (§ 10.4.3) 不能作为输出参数传递。

在函数成员调用 (§ 7.4.3) 的运行时处理期间，将按顺序从左到右计算参数列表的表达式或变量引用，具体规则如下：

- 对于值参数，计算参数表达式并执行到相应的参数类型的隐式转换 (§ 6.1)。结果值在函数成员调用中成为该值参数的初始值。
- 对于引用参数或输出参数，计算对应的变量引用，所得的存储位置在函数成员调用中成为该参数表示的存储位置。如果作为引用参数或输出参数给定的变量引用是一个引用类型的数组元素，则执行一个运行时检查以确保该数组的元素类型与参数的类型相同。如果检查失败，则引发 `System.ArrayTypeMismatchException`。

方法、索引器和实例构造函数可以将其最右边的参数声明为参数数组 (§ 10.5.1.4)。以正常形式还是以展开形式调用这类函数成员取决于哪种形式适用 (§ 7.4.2.1)：

- 当以正常形式调用带有参数数组的函数成员时，为参数数组给定的参数必须是属于某个类型的单个表达式，而该类型可隐式转换 (§ 6.1) 为参数数组类型。在此情况下，参数数组的作用与值参数完全一样。
- 当以展开形式调用带有参数数组的函数成员时，调用必须为参数数组指定零个或多个参数，其中每个参数都是某个类型的表达式，而该类型可隐式转换 (§ 6.1) 为参数数组的元素类型。在此情况下，调用会创建一个该参数数组类型的实例，

其所含的元素个数等于给定的参数个数，再用给定的参数值初始化此数组实例的每个元素，然后将新创建的数组实例用做实参。

对于参数列表的表达式总是按其书写的顺序来计算。因此，示例

```
class Test
{
    static void F(int x, int y, int z) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }
    static void Main() {
        int i = 0;
        F(i++, i++, i++);
    }
}
```

产生以下输出

```
x = 0, y = 1, z = 2
```

如果存在从 **B** 到 **A** 的隐式引用转换，则数组协方差规则 (§ 12.5) 允许数组类型 **A[]** 的值成为对数组类型 **B[]** 的一个实例的引用。由于这些规则，在将引用类型的数组元素作为引用参数或输出参数传递时，需要执行运行时检查以确保该数组的实际元素类型与参数的类型完全一致。在下面的示例中，第二个 **F** 调用导致引发 **System.ArrayTypeMismatchException**，原因是 **b** 的实际元素类型是 **string** 而不是 **object**。

```
class Test
{
    static void F(ref object x) {...}
    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);    // Ok
        F(ref b[1]);    // ArrayTypeMismatchException
    }
}
```

当以展开形式调用带有参数数组的函数成员时，其调用处理过程完全类似于如下过程：将带有数组初始值设定项 (§ 7.5.10.2) 的数组创建表达式插入到展开的参数所在之处。例如，已知下面的声明：

```
void F(int x, int y, params object[] args);
```

下列方法的展开形式的调用：

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

完全对应于：

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

请特别注意，当为参数数组指定的参数个数为零时，将创建一个空数组。

7.4.2 重载决策

重载决策是一种编译时机制，用于在给定了参数列表和一组候选函数成员的情况下，选择一个最佳函数成员来实施调用。在 C# 内，重载决策在下列不同的上下文中选择一个应调用的函数成员：

- 要调用的方法名称出现在调用表达式 (§ 7.5.5) 中。
- 要调用的实例构造函数出现在对象创建表达式 (§ 7.5.10.1) 中。
- 对索引器访问器的调用出现在元素访问 (§ 7.5.6) 中。
- 要调用的预定义运算符或用户定义的运算符出现在表达式 (§ 7.2.3 和 § 7.2.4) 中。

这些上下文中的每一个都以自己的惟一方式定义候选函数成员集和参数列表，上面列出的章节对此进行了详细说明。例如，方法调用的候选集不包括标记为 `override` (§ 7.3) 的方法，而且如果派生类中的任何方法可用 (§ 7.5.5.1)，则基类中的方法不是候选方法。

一旦确定了候选函数成员和参数列表，对最佳函数成员的选择在所有情况下都相同，都遵循下列规则。

如果给定了适用的候选函数成员集，则在其中选出最佳函数成员。如果该集合只包含一个函数成员，则该函数成员为最佳函数成员。否则，最佳函数成员的选择依据是各成员对给定的参数列表的匹配程度。比所有其他函数成员匹配得更好的那个函数成员就是最佳函数成员，但有一个前提：必须使用 § 7.4.2.2 中的规则将每个函数成员与所有其他函数成员进行比较。如果不是正好有一个函数成员比所有其他函数成员都好，则函数成员调用不明确并发生编译时错误。

下面几节定义有关术语“适用的函数成员”和“更好的函数成员”的准确含义。

7.4.2.1 适用的函数成员

当所有下列条件都为真时，就称函数成员对于参数列表 A 是一个适用的函数成员。

- A 中的参数数目与函数成员声明中的参数数目相同。
- 对于 A 中的每个参数，自变量的参数传递模式（值、`ref` 或 `out`）与相应参数的参数传递模式相同，而且：
 - 对于值参数或参数数组，存在从自变量类型到相应参数的类型的隐式转换 (§ 6.1)。
 - 对于 `ref` 或 `out` 参数，自变量的类型与相应参数的类型相同。`ref` 或 `out` 参数毕竟只是传递自变量的别名。

对于包含参数数组的函数成员，如果按上述规则判定该函数成员是适用的，则称它以正常形式适用。如果包含参数数组的函数成员以正常形式不适用，则该函数成员可能以展开形式适用。

- 构造展开形式的方法是：用参数数组的元素类型的零个或多个值参数替换函数成员声明中的参数数组，使参数列表 A 中的参数数目匹配总的参数数目。如果 A 中的自变量比函数成员声明中的固定参数的数目少，则该函数成员的展开形式无法构造，因而可判定该函数成员不适用。
- 如果声明函数成员类、结构或接口已经包含另一个与展开形式具有相同签名的

适用函数成员，则展开形式不适用。

- 否则，如果对于 A 中的每个自变量，它的参数传递模式与相应参数的参数传递模式相同，并且下列条件成立，则称该成员函数以展开形式适用。
 - 对于固定值参数或展开操作所创建的值参数，存在从自变量的类型到相应参数的类型的隐式转换（§ 6.1）；
 - 或者对于 ref 或 out 参数，参数的类型与相应参数的类型相同。

7.4.2.2 更好的函数成员

给定一个带有参数类型集 $\{A_1, A_2, \dots, A_N\}$ 的参数列表 A 和带有参数类型 $\{P_1, P_2, \dots, P_N\}$ 和 $\{Q_1, Q_2, \dots, Q_N\}$ 的两个可应用的函数成员 M_P 和 M_Q ，则在以下情况中， M_P 定义为比 M_Q 更好的函数成员：

- 对于每个参数，从 A_X 到 P_X 的隐式转换都不比从 A_X 到 Q_X 的隐式转换差，并且
- 对于至少一个参数，从 A_X 到 P_X 的转换比从 A_X 到 Q_X 的转换更好。

当执行此计算时，如果 M_P 或 M_Q 以展开形式适用，则 P_X 或 Q_X 所代表的是展开形式的参数列表中的参数。

7.4.2.3 更好的转换

假设有一个从类型 S 转换到类型 T_1 的隐式转换 C_1 ，以及一个从类型 S 转换到类型 T_2 的隐式转换 C_2 ，将按下列规则确定这两个转换中哪个是更好的转换。

- 如果 T_1 和 T_2 是相同的类型，则两个转换都不是更好的转换。
- 如果 S 为 T_1 ，则 C_1 为更好的转换。
- 如果 S 为 T_2 ，则 C_2 为更好的转换。
- 如果存在从 T_1 到 T_2 的隐式转换，且不存在从 T_2 到 T_1 的隐式转换，则 C_1 为更好的转换。
- 如果存在从 T_2 到 T_1 的隐式转换，且不存在从 T_1 到 T_2 的隐式转换，则 C_2 为更好的转换。
- 如果 T_1 为 sbyte 而 T_2 为 byte, ushort, uint 或 ulong，则 C_1 为更好的转换。
- 如果 T_2 为 sbyte 而 T_1 为 byte, ushort, uint 或 ulong，则 C_2 为更好的转换。
- 如果 T_1 为 short 而 T_2 为 ushort, uint 或 ulong，则 C_1 为更好的转换。
- 如果 T_2 为 short 而 T_1 为 ushort, uint 或 ulong，则 C_2 为更好的转换。
- 如果 T_1 为 int 而 T_2 为 uint 或 ulong，则 C_1 为更好的转换。
- 如果 T_2 为 int 而 T_1 为 uint 或 ulong，则 C_2 为更好的转换。
- 如果 T_1 为 long 而 T_2 为 ulong，则 C_1 为更好的转换。
- 如果 T_2 为 long 而 T_1 为 ulong，则 C_2 为更好的转换。
- 否则，两个转换都不是更好的转换。

如果按上述规则，确定了隐式转换 C_1 是比隐式转换 C_2 更好的转换，则 C_2 是比 C_1 更差的转换。

7.4.3 函数成员调用

本节描述在运行时发生的、调用一个特定的函数成员的进程。这里假定这个要调用的特定成员已在编译时进程确定了（可能采用重载决策从一组候选函数成员中选出）。

为了描述调用进程，将函数成员分成两类：

- 静态函数成员。包括实例构造函数、静态方法、静态属性访问器和用户定义的运算符。静态函数成员总是非虚拟的。
- 实例函数成员。包括实例方法、实例属性访问器和索引器访问器。实例函数成员不是非虚拟的就是虚拟的，并且总是在特定的实例上调用。该实例由实例表达式计算，并且在函数成员内可以以 `this` (§ 7.5.7) 的形式对它进行访问。

函数成员调用的运行时处理包括以下步骤（其中 `M` 是函数成员，如果 `M` 是实例成员，则 `E` 是实例表达式）。

- 如果 `M` 是静态函数成员，则：
 - ◆ 它的参数列表按照 § 7.4.1 中的说明进行计算。
 - ◆ 调用 `M`。
- 如果 `M` 是在值类型中声明的实例函数成员，则：
 - ◆ 计算 `E`。如果该计算导致异常，则不执行进一步的操作。
 - ◆ 如果 `E` 没有被归类为一个变量，则创建一个与 `E` 同类型的临时局部变量，并将 `E` 的值赋给该变量。这样，`E` 就被重新归类为对该临时局部变量的一个引用。该临时变量在 `M` 中可以以 `this` 的形式被访问，但不能以任何其他形式被访问。因此，仅当 `E` 是真正的变量时，调用方才可能观察到 `M` 对 `this` 所做的更改。
 - ◆ 参数列表按照 § 7.4.1 中的说明进行计算。
 - ◆ 调用 `M`。`E` 引用的变量成为 `this` 引用的变量。
- 如果 `M` 是在引用类型中声明的实例函数成员，则：
 - ◆ 计算 `E`。如果该计算导致异常，则不执行进一步的操作。
 - ◆ 参数列表按照 § 7.4.1 中的说明进行计算。
 - ◆ 如果 `E` 的类型为值类型，则执行装箱转换 (§ 4.3.1) 以将 `E` 转换为 `object` 类型，并且在下列步骤中，`E` 被视为 `object` 类型。这种情况下，`M` 只能是 `System.Object` 的成员。
 - ◆ 检查 `E` 的值是否有效。如果 `E` 的值为 `null`，则引发 `System.NullReferenceException`，并且不执行进一步的操作。
 - ◆ 要调用的函数成员实现按以下规则确定。
 - 如果 `E` 的编译时类型是接口，则调用的函数成员是 `M` 的实现，此实现是由 `E` 引用的实例在运行时所属的类型提供的。确定此函数成员时，应用接口映射规则 (§ 13.4.2) 确定由 `E` 引用的实例运行时类型提供的 `M` 实现。
 - 否则，如果 `M` 是虚函数成员，则调用的函数成员是由 `E` 引用的实例运行时类型提供的 `M` 实现。确定此函数成员时，应用“确定 `M` 的派生程度最

大的实现”的规则 (§ 10.5.3) (相对于 E 引用的实例的运行时类型)。

- 否则, M 是非虚函数成员, 调用的函数成员是 M 本身。

◆ 调用在上一步中确定的函数成员实现。E 引用的对象成为 this 引用的对象。

在下列情况中, 一个在值类型中实现的函数成员, 可以通过该值类型的已装箱实例来调用:

- 当该函数成员是从 object 类型继承的, 且具有 override 修饰符, 并通过 object 类型的实例表达式被调用时。
- 当函数成员是接口函数成员的实现并且通过接口类型的实例表达式被调用时。
- 当函数成员通过委托被调用时。

在上述情况下, 已装箱实例被视为包含值类型的一个变量, 并且此变量在函数成员调用中成为 this 引用的变量。具体说来, 这表示当调用已装箱实例的函数成员时, 该函数成员可以修改已装箱实例中包含的值。

7.5 基本表达式

基本表达式包括最简单的表达式形式。

primary-expression: (基本表达式:)

primary-no-array-creation-expression (非数组创建基本表达式)

array-creation-expression (数组创建表达式)

primary-no-array-creation-expression: (非数组创建基本表达式:)

literal (文本)

simple-name (简单名称)

parenthesized-expression (带括号的表达式)

member-access (成员访问)

invocation-expression (调用表达式)

element-access (元素访问)

this-access (this 访问)

base-access (base 访问)

post-increment-expression (后增量表达式)

post-decrement-expression (后减量表达式)

object-creation-expression (对象创建表达式)

delegate-creation-expression (委托创建表达式)

typeof-expression (typeof 表达式)

checked-expression (checked 表达式)

unchecked-expression (unchecked 表达式)

基本表达式分为数组创建表达式和非数组创建基本表达式。采用这种方式处理数组创建表达式 (不允许它和其他简单的表达式并列), 使语法能够禁止可能的代码混乱, 如:

```
object o = new int[3][1];
```


被另外解释为:

```
object o = (new int[3])[1];
```

7.5.1 文本

由文本 (§ 2.4.4) 组成的基本表达式属于值类别。

7.5.2 简单名称

简单名称由单个标识符组成。

simple-name: (简单名称:)

identifier (标识符)

简单名称按下面这样计算和分类。

- 如果简单名称在一个块内出现, 而且该块 (或封闭块) 的局部变量声明空间 (§ 3.3) 内含有一个以给定的简单名称命名的局部变量或参数, 则该简单名称引用该局部变量或参数并归为变量类别。
- 否则, 对于每个类型 *T* (此 *T* 从该简单名称的直接封闭类、结构或枚举声明开始, 从里到外, 遍历所有外部的封闭类或结构声明), 在 *T* 中对简单名称实施“成员查找”, 如果产生一个匹配, 则按下述内容进行:
 - ◆ 如果 *T* 为直接封闭类或结构类型, 且该成员查找标识了一个或多个方法, 则结果是一个具有关联实例表达式 *this* 的方法组。
 - ◆ 如果 *T* 为直接封闭类或结构类型, 查找标识出了一个实例成员, 并且引用在实例构造函数、实例方法或实例访问器的块中发生, 则结果与 *this.E* (其中 *E* 为简单名称) 形式的成员访问 (§ 7.5.4) 相同。
 - ◆ 否则, 结果与 *T.E* 形式 (其中 *E* 为简单名称) 的成员访问 (§ 7.5.4) 相同。在此情况下, 会因使用简单名称引用了实例成员而导致编译时错误。
- 否则, 从出现简单名称的命名空间开始, 然后是该命名空间外部的每个封闭命名空间 (若有), 最后以全局命名空间结束, 按下列步骤进行计算, 直到找到实体。
 - ◆ 如果命名空间包含有给定名称的命名空间成员, 则简单名称引用该成员, 并根据该成员归为命名空间或类型类别。
 - ◆ 如果该命名空间具有一个相应的命名空间声明, 将简单名称出现的位置包含在其中, 则:
 - 如果该命名空间声明包含一个 **using** 别名指令, 它使给定的简单名称与导入的命名空间或类型相关联, 则简单名称引用此命名空间或类型。
 - 如果该命名空间声明中包含多个 “**using** 命名空间” 指令, 而它们导入的命名空间只包含一个具有给定名称的类型, 则简单名称引用此类型。
 - 如果在上述导入的命名空间中, 包含有多个具有给定名称的类型, 则简单名称不明确并发生编译时错误。
- 否则, 由简单名称给定的名称被认为是未经定义的, 并导致编译时错误。

对于表达式中以简单名称形式给定的标识符的每个匹配项，直接封闭块（§ 8.2）或 switch 块（§ 8.7.2）内的表达式中同一标识符的每个其他匹配项都必须引用相同的实体。此规则确保了表达式上下文中的名称含义在一个块内始终相同。

示例

```
class Test
{
    double x;
    void F(bool b) {
        x = 1.0;
        if (b) {
            int x;
            x = 1;
        }
    }
}
```

将产生编译时错误，这是因为 x 引用外部块（其范围包括 if 语句中的嵌套块）中的不同实体。相反，示例

```
class Test
{
    double x;
    void F(bool b) {
        if (b) {
            x = 1.0;
        }
        else {
            int x ;
            x = 1;
        }
    }
}
```

是允许的，这是因为在外部块中从未使用过名称 x。

注意，固定含义的规则仅适用于简单名称。同一标识符在作为简单名称时有一种意义，在作为一个成员访问（§ 7.5.4）的右操作数时具有另一种意义，这是完全合法的。例如：

```
struct Point
{
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

这个示例阐释了一个将字段名用做实例构造函数中的参数名的通用模式。在该示例中，简单名称 x 和 y 引用参数，但这并不妨碍成员访问表达式 this.x 和 this.y 访问字段。

7.5.3 带括号的表达式

带括号的表达式由一个用括号括起来的表达式组成。

parenthesized-expression: (带括号的表达式:)

(expression) (表达式)

通过计算括号内的“表达式”计算带括号的表达式。如果括号内的表达式表示命名空间、类型或方法组，则发生编译时错误。否则，带括号的表达式的结果为所含表达式的计算结果。

7.5.4 成员访问

成员访问包括一个基本表达式或预定义类型，后跟“.”标记，再跟一个标识符。

member-access: (成员访问:)

primary-expression . identifier (基本表达式 . 标识符)

predefined-type . identifier (预定义类型 . 标识符)

predefined-type: one of (预定义类型: 下列之一)

bool byte char decimal double float int long

object sbyte short string uint ulong ushort

E.I 形式 (其中 E 是一个基本表达式或预定义类型, I 是标识符) 的成员访问按下面这样计算和分类。

- 如果 E 是命名空间, 而 I 是该命名空间的可访问成员的名称, 则结果为该成员, 并且根据该成员归为命名空间或类型。
- 如果 E 是一个预定义类型或一个被归类为类型的基本表达式, 并且 E 中的 I 成员查找 (§ 7.3) 产生一个匹配, 则 E.I 按下面这样计算和分类。
 - ◆ 如果 I 标识类型, 则结果为该类型。
 - ◆ 如果 I 标识一个或多个方法, 则结果为一个没有关联的实例表达式的方法组。
 - ◆ 如果 I 标识一个 static 属性, 则结果为一个没有关联的实例表达式的属性访问。
 - ◆ 如果 I 标识 static 字段, 则:
 - 如果该字段为 readonly 并且引用发生在声明该字段的类或结构的静态构造函数外, 则结果为值, 即 E 中静态字段 I 的值。
 - 否则, 结果为变量, 即 E 中的静态字段 I。
 - ◆ 如果 I 标识 static 事件, 则:
 - 如果引用发生在声明了该事件的类或结构内, 并且事件不是用事件访问器声明 (§ 10.7) 声明的, 则完全将 I 视为静态字段来处理 E.I。
 - 否则, 结果为没有关联的实例表达式的事件访问。
 - ◆ 如果 I 标识常数, 则结果为值, 即该常数的值。
 - ◆ 如果 I 标识枚举成员, 则结果为值, 即该枚举成员的值。
 - ◆ 否则, E.I 是无效成员引用, 并且发生编译时错误。
- 如果 E 为属性访问、索引器访问、变量或值, 设它们都属于类型为 T, 并且 T 中的 I 成员查找 (§ 7.3) 产生一个匹配, 则 E.I 按下面这样计算和分类。
 - ◆ 如果 E 为属性访问或索引器访问, 则获取属性访问或索引器访问的值 (§

7.1.1), 并且将 E 重新划为值类别。

- ◆ 如果 I 标识一个或多个方法, 则结果为具有 E 的关联实例表达式的方法组。
- ◆ 如果 I 标识实例属性, 则结果为具有 E 的关联实例表达式的属性访问。
- ◆ 如果 T 为类类型并且 I 标识此类类型的一个实例字段, 则:
 - 如果 E 的值为 null, 则引发 System.NullReferenceException。
 - 如果字段为 readonly 并且引用发生在声明字段的类的实例构造函数外, 则结果为值, 即 E 引用的对象中字段 I 的值。
 - 否则, 结果为变量, 即 E 引用的对象中的字段 I。
- ◆ 如果 T 为结构类型并且 I 标识此结构类型的实例字段, 则:
 - 如果 E 为值, 或者如果字段为 readonly 并且引用发生在声明字段的结构的实例构造函数外, 则结果为值, 即 E 给定的结构实例中字段 I 的值。
 - 否则, 结果为变量, 即 E 给定的结构实例中的字段 I。
- ◆ 如果 I 标识实例事件, 则:
 - 如果引用发生在声明事件的类或结构内, 并且事件不是用事件访问器声明 (§ 10.7) 声明的, 则完全将 I 视为实例字段来处理 E.I。
 - 否则, 结果为具有 E 的关联实例表达式的事件访问。

● 否则, E.I 是无效成员引用, 并且发生编译时错误。

在 E.I 形式的成员访问中, 如果 E 为单个标识符, E 可能有两种含义: 作为简单名称 (§ 7.5.2) 的 E, 作为类型名 (§ 3.8) 的 E。只要前者所标识的对象实体 (无论是常数、字段、属性、局部变量或参数) 所属的类型就是以后者命名的类型, 则 E 的这两种可能的含义都是允许的。在此规则下, E.I 可能有两种含义, 但它们永远是明确的, 因为在两种情况下, I 都必须是类型 E 的成员。换言之, 此规则在访问 E 的静态成员和嵌套类型时, 能简单地避免本来可能发生的编译时错误。例如:

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);
    public Color Complement() {...}
}
class A
{
    public Color Color;           // Color 类型的 Color 字段
    void F() {
        Color = Color.Black;     // 引用 Color.Black 静态成员
        Color = Color.Complement(); // 在 Color 字段上调用 Complement()
    }
    static void G() {
        Color c = Color.White;   // 引用 Color.White 静态成员
    }
}
```

在类 A 中, 引用 Color 类型的 Color 标识符的那些匹配项带下划线, 而引用 Color 字段的那些匹配项不带下划线。

7.5.5 调用表达式

调用表达式用于调用方法。

invocation-expression: (调用表达式:)

primary-expression (argument-list_{opt}) (基本表达式 (参数列表_{可选}))

调用表达式的基本表达式必须是方法组或委托类型的值。如果基本表达式是方法组，则调用表达式为方法调用 (§ 7.5.5.1)。如果基本表达式是委托类型的值，则调用表达式为委托调用 (§ 7.5.5.2)。如果基本表达式既不是方法组也不是委托类型的值，则发生编译时错误。

可选的参数列表 (§ 7.4.1) 列出的值或变量引用在调用时传递给方法中的参数。

调用表达式的计算结果按下面这样分类：

- 如果调用表达式调用的方法或委托返回 void，则结果为 Nothing。Nothing 类别的表达式不能是任何运算符的操作数，并且只能在语句表达式 (§ 8.6) 的上下文中使用。
- 否则，结果是由方法或委托返回的类型的值。

7.5.5.1 方法调用

对于方法调用，调用表达式的基本表达式必须是方法组。方法组标识要调用的一个方法，或者标识从中选择要调用的特定方法的一个重载方法集。在后一种情况中，具体调用哪个方法取决于参数列表中自变量的类型所提供的上下文。

M(A) 形式 (其中 M 是方法组，A 是可选的参数列表) 的方法调用的编译时处理包括以下步骤。

- 构造方法调用的候选方法集。从与 M 关联的、由以前的成员查找 (§ 7.3) 找到的方法集合开始，将该集合缩减为对于参数列表 A 适用的那些方法。集合缩减操作包括将下列规则应用于集合中的每个 T.N 方法 (其中 T 是声明方法 N 的类型):
 - 如果对于 A (§ 7.4.2.1)，N 不适用，则从集合中移除 N。
 - 如果对于 A (§ 7.4.2.1)，N 适用，则从集合中移除在 T 的基类型中声明的所有方法。
- 如果得到的候选方法集为空，则不存在适用的方法，并发生编译时错误。如果候选方法并非都声明为相同的类型，则方法调用不明确，并发生编译时错误 (只有对于具有多个直接基接口的接口中的方法调用才会出现后一种情况，详见 § 13.2.5 的介绍)。
- 使用 § 7.4.2 的重载决策规则确定候选方法集的最佳方法。如果无法确定单个最佳方法，则方法调用不明确，并发生编译时错误。
- 假设有一个最佳方法，方法调用在方法组的上下文中进行验证：如果最佳方法为静态方法，则方法组必须是由一个简单名称或一个通过类型的成员访问所产生的。如果最佳方法为实例方法，则方法组必须由简单名称、通过变量或值的成员访问或基访问产生。如果两个要求都不满足，则发生编译时错误。

通过以上步骤在编译时选定并验证了方法后，将根据 § 7.4.3 中说明的函数成员调用规则处理实际的运行时调用。

上述决策规则的直觉效果如下：为找到方法调用所调用的特定方法，从方法调用指示的类型开始，在继承链中一直向上查找，直到至少找到一个适用的、可访问的、非重写的方法声明。然后对该类型中声明的适用的、可访问的、非重写的方法集执行重载决策，并调用由此选定的方法。

7.5.5.2 委托调用

对于委托调用，调用表达式的基本表达式必须是委托类型的值。另外，将委托类型视为与委托类型具有相同的参数列表的函数成员，委托类型对于调用表达式的参数列表必须是适用的（§ 7.4.2.1）。

D(A) 形式（其中 D 是委托类型的基本表达式，A 是可选的参数列表）的委托调用的运行时处理包括以下步骤：

- 计算 D。如果此计算导致异常，则不执行进一步的操作。
- 检查 D 的值是否有效。如果 D 的值为 null，则引发 System.NullReferenceException，并且不再执行进一步的操作。
- 否则，D 是一个对委托实例的引用。对该委托的调用列表中的每个可调用实体上，执行函数成员调用（§ 7.4.3）。对于由实例和实例方法组成的可调用实体，用于调用的实例是包含在可调用实体中的实例。

7.5.6 元素访问

元素访问由一个“非数组创建基本表达式”，后面依次跟着“[”标记、表达式列表和“]”标记组成。表达式列表由一个或多个用逗号分隔的表达式组成。

element-access: (元素访问:)

primary-no-array-creation-expression [expression-list] (非数组创建基本表达式 [表达式列表])

expression-list: (表达式列表:)

expression (表达式)

expression-list , expression (表达式列表 , 表达式)

如果元素访问的非数组创建基本表达式是数组类型的值，则该元素访问是一个数组访问（§ 7.5.6.1）。否则，该非数组创建基本表达式必须是具有一个或多个索引器成员的类、结构或接口类型的变量或值，在这种情况下，元素访问为索引器访问（§ 7.5.6.2）。

7.5.6.1 数组访问

对于数组访问，元素访问的非数组创建基本表达式必须是数组类型的值。表达式列表中的表达式数目必须与数组类型的秩相同，并且每个表达式都必须是 int, uint, long, ulong 等类型，或者是可以隐式转换为这些类型中的一个或多个的类型。

数组访问的计算结果是数组的元素类型的变量，即由表达式列表中表达式的值选定的

数组元素。

$P[A]$ 形式（其中 P 是数组类型的非数组创建基本表达式， A 是表达式列表）的数组访问运行时处理包括以下步骤。

- 计算 P 。如果此计算导致异常，则不执行进一步的操作。
- 按从左到右的顺序计算表达式列表的索引表达式。计算每个索引表达式后，执行到下列类型之一的隐式转换（§ 6.1）：`int`，`uint`，`long`，`ulong`。选择此列表中第一个存在相应隐式转换的类型。例如，如果索引表达式是 `short` 类型，则执行到 `int` 的隐式转换，这是因为可以执行从 `short` 到 `int` 和从 `short` 到 `long` 的隐式转换。如果计算索引表达式或后面的隐式转换时导致异常，则不再进一步计算索引表达式，并且不再执行进一步的操作。
- 检查 P 的值是否有效。如果 P 的值为 `null`，则引发 `System.NullReferenceException`，并且不再执行进一步的操作。
- 针对由 P 引用的数组实例的每个维度的实际界限，检查表达式列表中每个表达式的值。如果一个或多个值超出了范围，则引发 `System.IndexOutOfRangeException`，并且不再执行进一步的操作。
- 计算由索引表达式给定的数组元素的位置，此位置将成为数组访问的结果。

7.5.6.2 索引器访问

对于索引器访问，元素访问的非数组创建基本表达式必须是类、结构或接口类型的变量或值，并且此类型必须实现了一个或多个对于元素访问的表达式列表适用的索引器。

$P[A]$ 形式（其中 P 是类、结构或接口类型 T 的一个非数组创建基本表达式， A 是表达式列表）的索引器访问编译时处理包括以下步骤。

- 构造由 T 提供的索引器集。该集合由 T 或 T 的基类型中的所有符合下列条件的索引器组成：它们不是经 `override` 声明的，并且在当前上下文中可以访问（§ 3.5）。
- 将该集合缩减为那些适用的并且不被其他索引器隐藏的索引器。对该集合中的每个索引器 $S.I$ （其中 S 为声明索引器 I 的类型）应用下列规则：
 - 如果对于 A （§ 7.4.2.1）， I 不适用，则从该集合中移除 I 。
 - 如果对于 A （§ 7.4.2.1）， I 适用，则从该集合中移除在 S 的基类型中声明的所有索引器。
- 如果候选索引器结果集为空，则不存在适用的索引器，并发生编译时错误。如果这些候选索引器不是在同一个类型内声明的，则索引器访问不明确，并发生编译时错误（只有对于具有多个直接基接口的接口实例的索引器访问才会发生后一种情况）。
- 使用 § 7.4.2 的重载决策规则确定候选索引器集的最佳索引器。如果无法确定单个最佳索引器，则索引器访问不明确，并发生编译时错误。
- 表达式列表的索引表达式按从左到右的顺序计算。索引器访问的处理结果是属于索引器访问类别的表达式。索引器访问表达式引用在上一步骤中确定的索引器，并具有 P 的关联实例表达式和 A 的关联参数列表。

根据索引器访问的使用上下文,索引器访问导致调用该索引器的 `get` 访问器或 `set` 访问器。如果索引器访问是赋值的目标,则调用 `set` 访问器以赋新值 (§ 7.13.1)。在其他所有情况中,调用 `get` 访问器以获取当前值 (§ 7.1)。

7.5.7 this 访问

`this` 访问由保留字 `this` 组成。

`this-access`: (`this` 访问:)

`this`

`this` 访问只能在实例构造函数、实例方法或实例访问器的“块”中使用。它具有下列含义之一:

- 当 `this` 在类的实例构造函数内的基本表达式中使用,它属于值类别。此时,该值的类型是出现该表达式的类,并且它的值就是对所构造的对象的引用。
- 当 `this` 在类的实例方法或实例访问器内的基本表达式中使用,它属于值类别。此时,该值的类型是出现该表达式的类,并且它的值就是对为其调用方法或访问器的对象的引用。
- 当 `this` 在结构的实例构造函数内的基本表达式中使用,它属于变量类别。该变量的类型是此表达式所在的结构,并且该变量表示的正是所构造的结构。结构实例构造函数的 `this` 变量的行为与结构类型的 `out` 参数完全一样,具体说来,这表示该变量在实例构造函数的每个执行路径中必须已明确赋值。
- 当 `this` 在结构的实例方法或实例访问器内的基本表达式中使用,它属于变量类别。该变量的类型是此表达式所在的结构,并且该变量表示的是为其调用方法或访问器的结构。在结构实例方法中,`this` 变量的行为与结构类型的 `ref` 参数完全一样。

在以上列出的上下文以外的上下文内的基本表达式中使用 `this` 是编译时错误。具体而言就是不能在静态方法、静态属性访问器中或字段声明的“变量初始值设定项”中引用 `this`。

7.5.8 base 访问

`base` 访问由保留字 `base`,后跟一个“.”标记和一个标识符,或后跟一个用方括号括起来的表达式列表组成:

`base-access`: (`base` 访问:)

`base` . `identifier` (`base` . 标识符)

`base` [`expression-list`] (`base` [表达式列表])

`base` 访问用于访问被当前类或结构中名称相似的成员隐藏的基类成员。`base` 访问只能在实例构造函数、实例方法或实例访问器的块中使用。当 `base.I` 出现在类或结构中时,`I` 必须表示该类或结构的基类的一个成员。同样,当 `base[E]` 出现在一个类中时,该类的基类中必须存在适用的索引器。

在编译时, `base.I` 和 `base[E]` 形式的 `base` 访问表达式完全等价于 `((B)this).I` 和 `((B)this)[E]` (其中 `B` 是所涉及的类或结构的基类), 计算方法是一样的。因此, `base.I` 和 `base[E]` 与 `this.I` 和 `this[E]` 相对应, 但 `this` 被视为基类的实例。

当某个 `base` 访问引用虚函数成员 (方法、属性或索引器) 时, 确定在运行时调用哪个函数成员的规则 (§ 7.4.3) 有一些更改。确定调用哪一个函数成员的方法是, 查找该函数成员相对于 `B` (而不是相对于 `this` 的运行时类型, 在非 `base` 访问中通常如此) 的派生程度最大的实现 (§ 10.5.3)。因此, 在 `virtual` 函数成员的 `override` 中, 可以使用 `base` 访问调用该函数成员的被继承了的实现。如果 `base` 访问引用的函数成员是抽象的, 则发生编译时错误。

7.5.9 后缀增量和后缀减量运算符

`post-increment-expression`: (后增量表达式:)

`primary-expression ++` (基本表达式 ++)

`post-decrement-expression`: (后减量表达式:)

`primary-expression --` (基本表达式 --)

后缀增量或后缀减量运算符的操作数必须是属于变量、属性访问或索引器访问类别的表达式。该运算的结果是与操作数类型相同的值。

如果后缀增量或后缀减量的操作数为属性或索引器访问, 则该属性或索引器必须同时具有 `get` 和 `set` 访问器。否则, 将发生编译时错误。

一元运算符重载决策 (§ 7.2.3) 适用于选择特定的运算符实现。对于下列类型存在着预定义的 `++` 和 `--` 运算符: `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 和任何枚举类型。预定义 `++` 运算符返回的结果值为操作数加上 1, 预定义 `--` 运算符返回的结果值为操作数减去 1。

`x++` 或 `x--` 形式的后缀增量或后缀减量运算的运行时处理包括以下步骤。

- 如果 `x` 属于变量:
 - 计算 `x` 以产生变量。
 - 保存 `x` 的值。
 - 调用选定的运算符, 将所保存的 `x` 值作为参数。
 - 运算符返回的值存储在由 `x` 的计算结果给定的位置中。
 - `x` 的保存值成为运算结果。
- 如果 `x` 属于属性或索引器访问:
 - 计算与 `x` 关联的实例表达式 (如果 `x` 不是 `static`) 和参数列表 (如果 `x` 是索引器访问), 结果用于后面的 `get` 和 `set` 访问器调用。
 - 调用 `x` 的 `get` 访问器并保存返回的值。
 - 调用选定的运算符, 将 `x` 的保存值作为参数。
 - 调用 `x` 的 `set` 访问器, 将运算符返回的值作为 `value` 参数。
 - `x` 的保存值成为运算结果。

`++` 和 `--` 运算符还支持前缀符 (§ 7.6.5)。 `x++` 或 `x--` 的结果是运算“之前” `x` 的

值，而 `++x` 或 `--x` 的结果是运算“之后”`x` 的值。在任何一种情况下，运算后 `x` 本身都具有相同的值。

`operator ++` 或 `operator --` 的实现既可以用后缀表示法调用，也可以用前缀表示法调用。但是，不能让这两种表示法分别去调用该运算符的不同实现。

7.5.10 new 运算符

`new` 运算符用于创建类型的新实例。

有三种形式的 `new` 表达式：

- 对象创建表达式用于创建类类型和值类型的新实例。
- 数组创建表达式用于创建数组类型的新实例。
- 委托创建表达式用于创建委托类型的新实例。

`new` 运算符表示创建类型的一个实例，但并不表示要为其动态分配内存。具体说来，值类型的实例不要求在表示它的变量以外有额外的内存，因而，在使用 `new` 创建值类型的实例时不发生动态分配内存。

7.5.10.1 对象创建表达式

对象创建表达式用于创建类类型或值类型的新实例。

object-creation-expression: (对象创建表达式:)

`new type (argument-listopt) (new 类型 (参数列表可选))`

对象创建表达式的类型必须是类类型或值类型。该类型不能是 `abstract` 类类型。

仅当类型为类类型或结构类型时才允许使用可选的参数列表 (§ 7.4.1)。

`new T(A)` 形式 (其中 `T` 是类类型或值类型，`A` 是可选参数列表) 的对象创建表达式的编译时处理包括以下步骤。

- 如果 `T` 是值类型且 `A` 不存在：
 - 对象创建表达式是默认构造函数调用。对象创建表达式的结果是 `T` 类型的一个值，即在 § 4.1.1 中定义的 `T` 的默认值。
- 如果 `T` 是类类型或结构类型：
 - 如果 `T` 为 `abstract` 类类型，则发生编译时错误。
 - 使用 § 7.4.2 的重载决策规则确定要调用的实例构造函数。候选实例构造函数集由所有在 `T` 中声明的可访问实例构造函数组成，这些构造函数对于 `A` (§ 7.4.2.1) 是适用的。如果候选实例构造函数集合为空，或者无法标识单个最佳实例构造函数，则发生编译时错误。
 - 对象创建表达式的结果是 `T` 类型的值，即由调用在上面的步骤中确定的实例构造函数所产生的值。
- 否则，对象创建表达式无效，并发生编译时错误。

`new T(A)` 形式 (其中 `T` 是类类型或结构类型，`A` 是可选参数列表) 的对象创建表达式的运行时处理包括以下步骤。

- 如果 `T` 是类类型：

- 为 T 类的一个新实例分配存储位置。如果没有足够的可用内存来分配新实例，则引发 `System.OutOfMemoryException`，并且不执行进一步的操作。
- 新实例的所有字段初始化为它们的默认值 (§ 5.2)。
- 根据函数成员调用 (§ 7.4.3) 的规则来调用实例构造函数。对新分配的实例的引用会自动传递给实例构造函数，因而，可以在实例构造函数中用 `this` 来访问该实例。
- 如果 T 是结构类型：
 - 通过分配一个临时局部变量来创建类型 T 的实例。由于要求结构类型的实例构造函数为所创建的实例的每个字段明确赋值，因此不需要初始化此临时变量。
 - 根据函数成员调用 (§ 7.4.3) 的规则来调用实例构造函数。对新分配的实例的引用会自动传递给实例构造函数，因而，可以在实例构造函数中用 `this` 来访问该实例。

7.5.10.2 数组创建表达式

数组创建表达式用于创建数组类型的新实例。

array-creation-expression: (数组创建表达式:)

`new non-array-type [expression-list] rank-specifiersopt`
`array-initializeropt (new 非数组类型 [表达式列表] 秩说明符可选 数组初始值设定项可选)`

`new array-type array-initializer (new 数组类型 数组初始值设定项)`

第一种形式的数组创建表达式分配一个数组实例，其类型是从表达式列表中删除每个表达式所得到的类型。例如，数组创建表达式 `new int[10, 20]` 产生 `int[,]` 类型的数组实例，数组创建表达式 `new int[10][,]` 产生 `int[,]` 类型的数组。表达式列表中的每个表达式必须属于 `int`、`uint`、`long` 或 `ulong` 类型，或者属于可以隐式转换为一种或多种这些类型的类型。每个表达式的值确定新分配的数组实例中相应维度的长度。由于数组维度的长度必须非负，因此，当表达式列表中出现带有负值的常数表达式时，将发生编译时错误。

除了在不安全上下文 (§ 18.1) 中外，数组的布局是未指定的。

如果第一种形式的数组创建表达式包含数组初始值设定项，则表达式列表中的每个表达式必须是常数，并且表达式列表指定的秩和维度长度必须匹配数组初始值设定项的秩和维度长度。

在第二种形式的数组创建表达式中，指定数组类型的秩必须匹配数组初始值设定项的秩。各维度长度通过数组初始值设定项的每个对应嵌套层数中的元素数来推断出。因此，表达式

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

完全对应于

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

对数组初始值设定项的介绍详见 § 12.6。

数组创建表达式的计算结果属于值类别，即对新分配的数组实例的一个引用。数组创

建表达式的运行时处理包括以下步骤。

- 表达式列表的维度长度表达式按从左到右的顺序计算。计算每个表达式之后，执行一个到下列类型之一的隐式转换 (§ 6.1)：int, uint, long, ulong。选择此列表中第一个存在相应隐式转换的类型。如果表达式计算或后面的隐式转换导致异常，则不计算其他表达式，并且不执行其他步骤。
- 维度长度的计算值按下面这样验证：如果一个或多个值小于零，则引发 System.OverflowException，并且不执行其他步骤。
- 分配具有给定维度长度的数组实例。如果没有足够的可用内存来分配新实例，则引发 System.OutOfMemoryException，并且不执行进一步的操作。
- 将新数组实例的所有元素初始化为它们的默认值 (§ 5.2)。
- 如果数组创建表达式包含数组初始值设定项，则计算数组初始值设定项中的每个表达式的值，并将该值赋值给它的相应数组元素。计算和赋值按数组初始值设定项中各表达式的写入顺序执行，换言之，按递增的索引顺序初始化元素，最右边的维度首先增加。如果给定表达式的计算或其面向相应数组元素的赋值导致异常，则不初始化其他元素（剩余的元素将因此具有它们的默认值）。

数组创建表达式允许实例化一个数组，并且它的元素也属于数组类型，但必须手动初始化这类数组的元素。例如，语句

```
int[][] a = new int[100][];
```

创建一个包含 100 个 int[] 类型的元素的一维数组。每个元素的初始值都为 null。想让数组创建表达式的同时也实例化它所指定的子数组是不可能的，因而，语句

```
int[][] a = new int[100][5];    // Error
```

会导致编译时错误。实例化子数组必须改为手动执行，如下所示

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

当数组的数组具有“矩形”形状时，即当子数组全都具有相同的长度时，使用多维数组更有效。在上面的示例中，实例化一个数组的数组时，实际上创建了 101 个对象（1 个外部数组和 100 个子数组）。相反，语句

```
int[,] = new int[100, 5];
```

只创建单个对象（一个二维数组）并在单个语句中完成分配。

7.5.10.3 委托创建表达式

委托创建表达式用于创建委托类型的新实例。

delegate-creation-expression: (委托创建表达式:)

```
new delegate-type ( expression ) (new 委托类型 ( 表达式 ))
```

委托创建表达式的参数必须是方法组 (§ 7.1) 或委托类型的值。如果参数是方法组，则它标识方法和（对于实例方法）为其创建委托的对象。如果参数是委托类型的值，则它标识为其创建副本的委托实例。

`new D(E)` 形式（其中 `D` 是委托类型，`E` 是表达式）的委托创建表达式的编译时处理包括以下步骤。

- 如果 `E` 是方法组：
 - 由 `E` 标识的方法集必须正好包括一个与 `D` 兼容的方法，并且此方法成为新创建的委托（§ 15.1）引用的方法。如果不存在匹配的方法，或者如果存在多个匹配的方法，则发生编译时错误。如果选定的方法是实例方法，则与 `E` 关联的实例表达式确定委托的目标对象。
 - 与方法调用中一样，选定的方法也必须与方法组的上下文兼容。如果该方法是静态方法，则方法组必须是由一个简单名称或关于类型的一个成员访问产生的。如果该方法是实例方法，则方法组必须是由一个简单名称或关于变量或值的一个成员访问产生的。如果选定的方法与方法组的上下文不匹配，则发生编译时错误。
 - 结果是类型 `D` 的值，即一个引用选定方法和目标对象的新创建的委托。
- 如果 `E` 是委托类型的值：
 - `D` 和 `E` 必须兼容，否则将发生编译时错误。
 - 结果是类型 `D` 的值，即引用与 `E` 具有相同调用列表的新创建的委托。
- 否则，委托创建表达式无效，并发生编译时错误。

`new D(E)` 形式（其中 `D` 是委托类型，`E` 是表达式）的委托创建表达式的运行时处理包括以下步骤。

- 如果 `E` 是方法组：
 - ◆ 如果在编译时选择的方法是静态方法，则委托的目标对象为 `null`。否则，选定的方法是实例方法，并且从 `E` 的关联实例表达式确定委托的目标对象。
 - 计算实例表达式。如果此计算导致异常，则不执行进一步的操作。
 - 如果实例表达式为引用类型，则由实例表达式计算的值成为目标对象。如果目标对象为 `null`，则引发 `System.NullReferenceException` 并且不执行其他步骤。
 - 如果实例表达式为值类型，则执行装箱操作（§ 4.3.1）以将值转换为对象，并且此对象成为目标对象。
 - ◆ 为委托类型 `D` 的一个新实例分配存储位置。如果没有足够的可用内存来为新实例分配存储位置，则引发 `System.OutOfMemoryException`，并且不执行进一步的操作。
 - ◆ 用对在编译时确定的方法的引用和对上面计算的目标对象的引用初始化新委托实例。
- 如果 `E` 是委托类型的值：
 - ◆ 计算 `E`。如果此计算导致异常，则不执行进一步的操作。
 - ◆ 如果 `E` 的值为 `null`，则引发 `System.NullReferenceException`，并且不执行进一步的操作。
 - ◆ 分配委托类型 `D` 的新实例。如果没有足够的可用内存来分配新实例，则引发 `System.OutOfMemoryException`，并且不执行进一步的操作。

◆ 用与 E 给定的委托实例相同的调用列表初始化新委托实例。

委托的调用列表在实例化委托时确定并在委托的整个生存期期间保持不变。换句话说，一旦创建了委托，就不可能更改它的可调用目标实体。当组合两个委托或从一个委托中移除另一个委托 (§ 15.1) 时，将产生新委托，现有委托的内容不更改。

不可能创建引用属性、索引器、用户定义的运算符、实例构造函数、析构函数或静态构造函数的委托。

如上所述，当从方法组创建一个委托时，需根据该委托的形参表和返回类型来确定要选择的重载方法。在下面的示例中，

```
delegate double DoubleFunc(double x);
class A
{
    DoubleFunc f = new DoubleFunc(Square);
    static float Square(float x) {
        return x * x;
    }
    static double Square(double x) {
        return x * x;
    }
}
```

A.f 字段将由引用第二个 **Square** 方法的委托初始化，因为该方法与 **DoubleFunc** 的形参表和返回类型完全匹配。如果第二个 **Square** 方法不存在，则发生编译时错误。

7.5.11 typeof 运算符

typeof 运算符用于获取类型的 **System.Type** 对象。

typeof-expression: (**typeof** 表达式:)

typeof (**type**) (**typeof** (**类型**))

typeof (**void**)

typeof 表达式的第一种形式由 **typeof** 关键字，后跟带括号的“类型”组成。这种形式的表达式的结果是与给定的类型对应的 **System.Type** 对象。任何给定的类型都只有一个 **System.Type** 对象。这意味着对类型 **T** 而言，**typeof(T) == typeof(T)** 始终为 **true**。

typeof 表达式的第二种形式由 **typeof** 关键字，后跟带括号的 **void** 关键字组成。这种形式的表达式的结果是一个表示“类型不存在”的 **System.Type** 对象。这种通过 **typeof(void)** 返回的类型对象与为所有类型返回的类型对象截然不同。这种特殊的类型对象在这样的类库中很有用：它允许在源语言中仔细考虑一些方法，希望可以用 **System.Type** 的实例来表示任何方法（包括 **void** 方法）的返回类型。

示例

```
using System;
class Test
{
    static void Main() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
        }
```



```

        typeof(string),
        typeof(double[]),
        typeof(void)
    };
    for (int i = 0; i < t.Length; i++) {
        Console.WriteLine(t[i].FullName);
    }
}
}

```

产生下列输出：

```

System.Int32
System.Int32
System.String
System.Double[]
System.Void

```

注意，`int` 和 `System.Int32` 是相同的类型。

7.5.12 checked 和 unchecked 运算符

`checked` 和 `unchecked` 运算符用于控制上下文中是否实施关于整型算术运算和转换的溢出检查。

checked-expression: (checked 表达式:)

`checked (expression) (checked (表达式))`

unchecked-expression: (unchecked 表达式:)

`unchecked (expression) (unchecked (表达式))`

`checked` 运算符在 `checked` 上下文中计算所包含的表达式，`unchecked` 运算符在 `unchecked` 上下文中计算所包含的表达式。除了在给定的溢出检查上下文中计算所包含的表达式外，`checked` 表达式或 `unchecked` 表达式与带括号的表达式 (§ 7.5.3) 完全对应。

也可以通过 `checked` 语句和 `unchecked` 语句 (§ 8.11) 控制溢出检查上下文。

下列运算符受由 `checked` 和 `unchecked` 运算符和语句所确定的溢出检查上下文影响。

- 预定义的 `++` 和 `--` 一元运算符 (§ 7.5.9) 和 (§ 7.6.5) (当操作数为整型时)。
- 预定义的一元运算符 `-` (§ 7.6.2) (当操作数为整型时)。
- 预定义的二元运算符 `+`, `-`, `*` 和 `/` (§ 7.7) (当两个操作数均为整型时)。
- 从一个整型到另一个整型或从 `float` 或 `double` 到整型的显式数值转换 (§ 6.2.1)。

当上面的运算之一产生的结果太大，无法用目标类型表示时，执行运算的上下文控制由此引起的行为。

- 在 `checked` 上下文中，如果运算发生在一个常数表达式 (§ 7.15) 中，则发生编译时错误。否则，当在运行时执行运算时，引发 `System.OverflowException`。
- 在 `unchecked` 上下文中，计算的结果被截断，放弃不适合目标类型的任何高序位。

对于不用任何 `checked` 或 `unchecked` 运算符或语句括起来的非常数表达式 (在运行时计算的表达式)，除非外部因素 (如编译器开关和执行环境配置) 要求 `checked` 计算，否

则溢出检查上下文默认为 `unchecked`。

对于常数表达式（可以在编译时完全计算的表达式），溢出检查上下文总是默认为 `checked`。除非将常数表达式显式放置在 `unchecked` 上下文中，否则在表达式的编译时计算期间发生的溢出总是导致编译时错误。

下面是一个示例：

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;
    static int F() {
        return checked(x * y);    // 引发 OverflowException
    }
    static int G() {
        return unchecked(x * y); // 返回-727 379 968
    }
    static int H() {
        return x * y;            // 取决于默认值
    }
}
```

其中，由于在编译时没有表达式可以计算，所以不报告编译时错误。在运行时，F 方法引发 `System.OverflowException`，G 方法返回 `-727 379 968`（从超出范围的结果中取较低的 32 位）。H 方法的行为取决于编译时设定的默认溢出检查上下文，但它不是与 F 相同就是与 G 相同。

下面是一个示例：

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;
    static int F() {
        return checked(x * y);    // 链接错误，溢出
    }
    static int G() {
        return unchecked(x * y); // 返回 -727 379 968
    }
    static int H() {
        return x * y;            // 链接错误，溢出
    }
}
```

其中，在计算 F 和 H 中的常数表达式时发生的溢出导致报告编译时错误，原因是表达式是在 `checked` 上下文中计算的。在计算 G 中的常数表达式时也发生溢出，但由于计算是在 `unchecked` 上下文中发生的，所以不报告溢出。

`checked` 和 `unchecked` 运算符只影响原文包含在“(”和“)”标记中的那些运算的溢出检查上下文。这些运算符不影响因计算包含的表达式而调用的函数成员。在下面的示例中，在 F 中使用 `checked` 不影响 `Multiply` 中的 `x * y` 计算，因此在默认溢出检查上下文中计算 `x * y`。

```
class Test
{
```

```

static int Multiply(int x, int y) {
    return x * y;
}
static int F() {
    return checked(Multiply(1000000, 1000000));
}
}

```

当以十六进制表示法编写有符号整型的常数时，`unchecked` 运算符很方便。例如：

```

class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);
    public const int HighBit = unchecked((int)0x80000000);
}

```

上面的两个十六进制常数均为 `uint` 类型。因为这些常数超出了 `int` 范围，所以如果不使用 `unchecked` 运算符，强制转换到 `int` 将产生编译时错误。

`checked` 和 `unchecked` 运算符和语句允许程序员控制一些数值计算的某些方面。当然，某些数值运算符的行为取决于其操作数的数据类型。例如，两个小数相乘总是导致溢出异常，即使是在显式 `unchecked` 结构内也是如此。同样，两个浮点数相乘从不会导致溢出异常，即使是在显式 `checked` 结构内也是如此。另外，其他运算符从不受检查模式（不管是默认的还是显式的）的影响。

7.6 一元运算符

`+`, `-`, `!`, `~`, `++`, `--` 和强制转换运算符称为一元运算符。

unary-expression: (一元表达式:)

primary-expression (基本表达式)

+ unary-expression (+ 一元表达式)

- unary-expression (- 一元表达式)

! unary-expression (! 一元表达式)

~ unary-expression (~ 一元表达式)

pre-increment-expression (前缀增量表达式)

pre-decrement-expression (前缀减量表达式)

cast-expression (强制转换表达式)

7.6.1 一元加运算符

对于 `+x` 形式的运算，应用一元运算符重载决策 (§ 7.2.3) 以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果类型是该运算符的返回类型。预定义的一元加运算符为：

```

int operator +(int x);
uint operator +(uint x);
long operator +(long x);

```

```
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

对于这些运算符，结果只是操作数的值。

7.6.2 一元减运算符

对于 $-x$ 形式的运算，应用一元运算符重载决策（§ 7.2.3）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果类型是该运算符的返回类型。预定义的否定运算符为：

- 整数否定：

```
int operator -(int x);
long operator -(long x);
```

通过从 0 中减去 x 来计算结果。如果 x 的值是操作数类型的最小可表示值（对 `int` 是 -2^{31} ，对 `long` 是 -2^{63} ），则 x 的算术否定在操作数类型中不可表示。如果这种情况发生在 `checked` 上下文中，则引发 `System.OverflowException`；如果发生在 `unchecked` 上下文中，则结果是操作数的值而且不报告溢出。

如果否定运算符的操作数为 `uint` 类型，则它转换为 `long` 类型，并且结果的类型为 `long`。例外是允许将 `int` 值 $-2\,147\,483\,648$ (-2^{31}) 写为十进制整数（§ 2.4.4.2）。

如果否定运算符的操作数为 `ulong` 类型，则发生编译时错误。例外是允许将 `long` 值 $-9\,223\,372\,036\,854\,775\,808$ (-2^{63}) 写为十进制整数（§ 2.4.4.2）。

- 浮点否定：

```
float operator -(float x);
double operator -(double x);
```

结果是符号被反转的 x 的值。如果 x 为 `NaN`，则结果也为 `NaN`。

- 小数否定：

```
decimal operator -(decimal x);
```

通过从 0 中减去 x 来计算结果。小数否定等效于使用 `System.Decimal` 类型的一元减运算符。

7.6.3 逻辑否定运算符

对于 $!x$ 形式的运算，应用一元运算符重载决策（§ 7.2.3）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果类型是该运算符的返回类型。只存在一个预定义的逻辑否定运算符：

```
bool operator !(bool x);
```

此运算符计算操作数的逻辑否定：如果操作数为 `true`，则结果为 `false`；如果操作数为 `false`，则结果为 `true`。

7.6.4 按位求补运算符

对于 $\sim x$ 形式的运算，应用一元运算符重载决策 (§ 7.2.3) 以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果类型是该运算符的返回类型。预定义的按位求补运算符为：

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

对于这些运算符，运算结果为 x 的按位求补。

每个 E 枚举类型都隐式地提供下列按位求补运算符：

```
E operator ~(E x);
```

$\sim x$ （其中 x 是具有基础类型 U 的枚举类型 E 的表达式）的计算结果与 $(E)(\sim(U)x)$ 的计算结果完全相同。

7.6.5 前缀增量和减量运算符

pre-increment-expression: (前缀增量表达式:)

++ unary-expression (**++** 一元表达式)

pre-decrement-expression: (前缀减量表达式:)

-- unary-expression (**--** 一元表达式)

前缀增量或前缀减量运算的操作数必须是属于变量、属性访问或索引器访问类别的表达式。该运算的结果是与操作数类型相同的值。

如果前缀增量或前缀减量运算的操作数是属性或索引器访问，则属性或索引器必须同时具有 `get` 和 `set` 访问器。否则，将发生编译时错误。

一元运算符重载决策 (§ 7.2.3) 用于选择一个特定的运算符实现。存在着适用于下列类型的预定义 **++** 和 **--** 运算符：`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` 和任何枚举类型。预定义 **++** 运算符返回的结果值为操作数加上 1，预定义 **--** 运算符返回的结果值为操作数减去 1。

++x 或 **--x** 形式的前缀增量或减量运算的运行时处理包括以下步骤。

- 如果 x 属于变量：
 - 计算 x 以产生变量。
 - 调用选定的运算符，将 x 的值作为参数。
 - 运算符返回的值存储在由 x 的计算结果给定的位置。
 - 运算符返回的值成为该运算的结果。
- 如果 x 属于属性或索引器访问：
 - 计算与 x 关联的实例表达式（如果 x 不是 `static`）和参数列表（如果 x 是索引器访问），结果用于后面的 `get` 和 `set` 访问器调用。

- 调用 `x` 的 `get` 访问器。
- 调用选定的运算符，将 `get` 访问器返回的值作为参数。
- 调用 `x` 的 `set` 访问器，将运算符返回的值作为 `value` 参数。
- 运算符返回的值成为该运算的结果。

`++` 和 `--` 运算符也支持后缀表示法 (§ 7.5.9)。`x++` 或 `x--` 的结果是运算“之前”`x` 的值，而 `++x` 或 `--x` 的结果是运算“之后”`x` 的值。在任何一种情况下，运算后 `x` 本身都具有相同的值。

`operator ++` 或 `operator --` 的实现既可以用后缀表示法调用，也可以用前缀表示法调用。但是，不能让这两种表示法分别去调用该运算符的不同实现。

7.6.6 强制转换表达式

强制转换表达式用于将表达式显式转换为给定类型。

cast-expression: (强制转换表达式:)

(type) unary-expression ((类型) 一元表达式)

(T)E 形式 (其中 `T` 是“类型”，`E` 是一元表达式) 的强制转换表达式执行把 `E` 的值转换到类型 `T` 的显式转换 (§ 6.2)。如果不存在从 `E` 的类型到 `T` 的显式转换，则发生编译时错误。否则，结果为显式转换产生的值。即使 `E` 表示变量，结果也总是为值类别。

强制转换表达式的文法表示可能导致某些语法多义性。例如，表达式 `(x)-y` 既可以按强制转换表达式解释 (`-y` 到类型 `x` 的强制转换)，也可以按结合了带括号的表达式的增量表达式解释 (计算 `x - y` 的值)。

下列规则用于解决强制转换表达式的多义性。仅当下列条件至少有一个为真时，在括号中的由一个或多个标记 (§ 2.4) 排列起来的序列才被视为强制转换表达式的开始。

- 标记的序列是关于类型的正确的文法表示，但对于表达式则不是。
- 标记的序列是关于类型的正确的文法表示，而且紧跟在右括号后面的标记是标记“~”、标记“!”、标记“(”、标识符 (§ 2.4.1)、文本 (§ 2.4.4) 或除 `as` 和 `is` 外的任何关键字 (§ 2.4.3)。

上面出现的术语“正确的文法表示”仅指标记的序列必须符合特定的文法产生式。它并没有特别考虑任何构成标识符的实际含义。例如，如果 `x` 和 `y` 是标识符，则 `x.y` 对于类型是正确的文法表示，即使 `x.y` 实际并不表示类型也是如此。

从上述消除歧义规则可以导出下述结论：如果 `x` 和 `y` 是标识符，则 `(x)y`，`(x)(y)` 和 `(x)(-y)` 为强制转换表达式，但 `(x)-y` 不是，即使 `x` 标识的是类型也是如此。然而，如果 `x` 是一个标识预定义类型 (如 `int`) 的关键字，则所有 4 种形式均为强制转换表达式 (因为这种关键字本身不可能是表达式)。

7.7 算术运算符

`*`，`/`，`%`，`+` 和 `-` 运算符称为算术运算符。

multiplicative-expression: (乘法表达式:)
unary-expression (一元表达式)
multiplicative-expression * unary-expression (乘法表达式 * 一元表达式)
multiplicative-expression / unary-expression (乘法表达式 / 一元表达式)
multiplicative-expression % unary-expression (乘法表达式 % 一元表达式)
additive-expression: (增量表达式:)
multiplicative-expression (乘法表达式)
additive-expression + multiplicative-expression (加法表达式 + 乘法表达式)
additive-expression - multiplicative-expression (加法表达式 - 乘法表达式)

7.7.1 乘法运算符

对于 $x * y$ 形式的运算，应用二元运算符重载决策 (§ 7.2.4) 以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果的类型是该运算符的返回类型。
下面列出了预定义的乘法运算符。这些运算符均计算 x 和 y 的乘积。

● 整数乘法:

```
int operator *(int x, int y);  
uint operator *(uint x, uint y);  
long operator *(long x, long y);  
ulong operator *(ulong x, ulong y);
```

在 checked 上下文中，如果乘积超出结果类型的范围，则引发 System.OverflowException。在 unchecked 上下文中，不报告溢出并且结果类型范围外的任何有效高序位都被放弃。

● 浮点乘法:

```
float operator *(float x, float y);  
double operator *(double x, double y);
```

根据 IEEE 754 算法法则计算乘积。下表列出了非零有限值、零、无限值和 NaN 的所有可能的组合结果。在表中， x 和 y 是正有限值， z 是 $x * y$ 的结果。如果结果对目标类型而言太大，则 z 为无穷大；如果结果对目标类型而言太小，则 z 为零。

	+y	-y	+0	-0	+∞	∞	NaN
+x	+z	-z	+0	-0	+∞	∞	NaN
-x	-z	+z	-0	+0	∞	+∞	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+∞	+∞	∞	NaN	NaN	+∞	∞	NaN
∞	∞	+∞	NaN	NaN	∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

● 小数乘法:

```
decimal operator *(decimal x, decimal y);
```

如果结果值太大，不能用 decimal 格式表示，则引发 System.OverflowException。
如果结果值太小，不能用 decimal 格式表示，则结果为零。在进行任何舍入之前，
结果的小数位数是两个操作数的小数位数的和。
小数乘法等效于使用 System.Decimal 类型的乘法运算符。

7.7.2 除法运算符

对于 x / y 形式的运算，应用二元运算符重载决策（§ 7.2.4）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果的类型是该运算符的返回类型。
下面列出了预定义的除法运算符。这些运算符均计算 x 和 y 的商。

● 整数除法:

```
int operator /(int x, int y);  
uint operator /(uint x, uint y);  
long operator /(long x, long y);  
ulong operator /(ulong x, ulong y);
```

如果右操作数的值为零，则引发 System.DivideByZeroException。
除法将结果舍入到零，并且结果的绝对值是小于两个操作数的商的绝对值的最大可能整数。当两个操作数符号相同时，结果为零或正；当两个操作数符号相反时，结果为零或负。
如果左操作数为最小可表示 int 或 long 值，右操作数为 -1，则发生溢出。无论操作是在 checked 还是在 unchecked 上下文中发生，此时总是引发 System.OverflowException。

● 浮点除法:

```
float operator /(float x, float y);  
double operator /(double x, double y);
```

根据 IEEE 754 算法法则计算商。下表列出了非零的有限值、零、无穷大和 NaN 的所有可能的组合结果。在表中，x 和 y 是正有限值，z 是 x / y 的结果。如果结果对目标类型而言太大，则 z 为无穷大。如果结果对目标类型而言太小，则 z 为零。

	+y	-y	+0	-0	+∞	∞	NaN
+x	+z	-z	+∞	∞	+0	-0	NaN
-x	-z	+z	∞	+∞	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+∞	+∞	∞	+∞	∞	NaN	NaN	NaN
∞	∞	+∞	∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

● 小数除法:

```
decimal operator /(decimal x, decimal y);
```

如果右操作数的值为零，则引发 `System.DivideByZeroException`。如果结果值太大，不能用 `decimal` 格式表示，则引发 `System.OverflowException`。如果结果值太小，不能用 `decimal` 格式表示，则结果为零。结果的小数位数是最小的小数位数，它保留等于最接近真实算术结果的可表示小数值的结果。
小数除法等效于使用 `System.Decimal` 类型的除法运算符。

7.7.3 余数运算符

对于 `x % y` 形式的运算，应用二元运算符重载决策 (§ 7.2.4) 以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果的类型是该运算符的返回类型。
下面列出了预定义的余数运算符。这些运算符均计算 `x` 除以 `y` 的余数。

● 整数余数:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

`x % y` 的结果是表达式 `x - (x / y) * y` 的值。如果 `y` 为零，则引发 `System.DivideByZeroException`。余数运算符从不导致溢出。

● 浮点余数:

```
float operator %(float x, float y);
double operator %(double x, double y);
```

下表列出了非零的有限值、零、无穷大和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 是正有限值，`z` 是 `x % y` 的结果并按 `x - n * y` (其中 `n` 是小于或等于 `x / y` 的最大可能整数) 计算。这种计算余数的方法类似于用于整数操作数的方法，但不同于 IEEE 754 定义 (在此定义中，`n` 是最接近 `x / y` 的整数)。

	+y	-y	+0	-0	+∞	∞	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

● 小数余数:

```
decimal operator %(decimal x, decimal y);
```

如果右操作数的值为零，则引发 `System.DivideByZeroException`。在进行任何舍入之前，结果的小数位数是两个操作数中较大的小数位数，而且结果的符号与 `x` 的相同（如果非零）。
小数余数等效于使用 `System.Decimal` 类型的余数运算符。

7.7.4 加法运算符

对于 `x + y` 形式的运算，应用二元运算符重载决策（§ 7.2.4）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果的类型是该运算符的返回类型。
下面列出了预定义的加法运算符。对于数值和枚举类型，预定义的加法运算符计算两个操作数的和。当一个或两个操作数为 `string` 类型时，预定义的加法运算符把两个操作数的字符串表示形式串联起来。

● 整数加法：

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

在 `checked` 上下文中，如果和超出结果类型的范围，则引发 `System.OverflowException`。在 `unchecked` 上下文中，不报告溢出，并且结果类型范围外的任何有效高序位都被放弃。

● 浮点加法：

```
float operator +(float x, float y);
double operator +(double x, double y);
```

根据 IEEE 754 算法法则计算和。下表列出了非零有限值、零、无限值和 NaN 的所有可能组合的结果。在表中，`x` 和 `y` 是非零的有限值，`z` 是 `x + y` 的结果。如果 `x` 和 `y` 的绝对值相同但符号相反，则 `z` 为正零。如果 `x + y` 太大，不能用目标类型表示，则 `z` 是与 `x + y` 具有相同符号的无穷大。

	y	+0	-0	+∞	∞	NaN
x	z	x	x	+∞	∞	NaN
+0	y	+0	+0	+∞	∞	NaN
-0	y	+0	-0	+∞	∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
∞	∞	∞	∞	NaN	∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

● 小数加法：

```
decimal operator +(decimal x, decimal y);
```

如果结果值太大，不能用 `decimal` 格式表示，则引发 `System.OverflowException`。在进行任何舍入之前，结果的小数位数是两个操作数中较大的小数位数。

小数加法等效于使用 `System.Decimal` 类型的加法运算符。

- 枚举加法。每个枚举类型都隐式提供下列预定义运算符，其中 `E` 为枚举类型，`U` 为 `E` 的基础类型：

```
E operator +(E x, U y);
E operator +(U x, E y);
```

这些运算符严格按 $(E)((U)x + (U)y)$ 计算。

- 字符串串联：

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

当一个或两个操作数为 `string` 类型时，二元 `+` 运算符执行字符串串联。在字符串串联运算中，如果它的一个操作数为 `null`，则用空字符串来替换此操作数。否则，任何非字符串参数都通过调用从 `object` 类型继承的虚 `ToString` 方法，转换为它的字符串表示形式。如果 `ToString` 返回 `null`，则替换成空字符串。

```
using System;
class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<"); // 显示 s = ><
        int i = 1;
        Console.WriteLine("i = " + i);        //显示 i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);        //显示 f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);        //显示 d = 2.900
    }
}
```

字符串串联运算符的结果是一个字符串，由左操作数的字符后跟右操作数的字符组成。字符串串联运算符从不返回 `null` 值。如果没有足够的内存可用于分配结果字符串，则可能引发 `System.OutOfMemoryException`。

- 委托组合。每个委托类型都隐式提供以下预定义运算符，其中 `D` 是委托类型：

```
D operator +(D x, D y);
```

当两个操作数均为某个委托类型 `D` 时，二元 `+` 运算符执行委托组合（如果操作数具有不同的委托类型，则发生编译时错误）。如果第一个操作数为 `null`，则运算结果为第二个操作数的值（即使此操作数也为 `null`）。否则，如果第二个操作数为 `null`，则运算结果为第一个操作数的值。否则，运算结果是一个新委托实例，该实例在被调用时调用第一个操作数，然后调用第二个操作数。有关委托组合的示例，请参见 § 7.7.5 和 § 15.3。由于 `System.Delegate` 不是委托类型，因此不为它定义 `operator +`。

7.7.5 减法运算符

对于 $x - y$ 形式的运算，应用二元运算符重载决策（§ 7.2.4）以选择特定的运算符实现。操作数转换为所选运算符的参数类型，结果的类型是该运算符的返回类型。

下面列出了预定义的减法运算符。这些运算符均从 x 中减去 y 。

● 整数减法：

```
int operator -(int x, int y);
uint operator -(uint x, uint y);
long operator -(long x, long y);
ulong operator -(ulong x, ulong y);
```

在 `checked` 上下文中，如果差超出结果类型的范围，则引发 `System.OverflowException`。在 `unchecked` 上下文中，不报告溢出并且结果类型范围外的任何有效高序位都被放弃。

● 浮点减法：

```
float operator -(float x, float y);
double operator -(double x, double y);
```

根据 IEEE 754 算术法则计算差。下表列出了非零的有限值、零、无穷大和 NaN 的所有可能组合的结果。在表中， x 和 y 是非零的有限值， z 是 $x - y$ 的结果。如果 x 和 y 相等，则 z 为正零。如果 $x - y$ 太大，不能用目标类型表示，则 z 是与 $x - y$ 具有相同符号的无穷大。

	y	$+0$	-0	$+\infty$	∞	NaN
x	z	x	x	∞	$+\infty$	NaN
$+0$	$-y$	$+0$	$+0$	∞	$+\infty$	NaN
-0	$-y$	-0	$+0$	∞	$+\infty$	NaN
$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN	$+\infty$	NaN
∞	∞	∞	∞	∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

● 小数减法：

```
decimal operator ~(decimal x, decimal y);
```

如果结果值太大，不能用 `decimal` 格式表示，则引发 `System.OverflowException`。在进行任何舍入之前，结果的小数位数是两个操作数中较大的小数位数。小数减法等效于使用 `System.Decimal` 类型的减法运算符。

● 枚举减法。每个枚举类型都隐式提供下列预定义运算符，其中 E 为枚举类型， U 为 E 的基础类型：

```
U operator -(E x, E y);
```

该运算符严格按 $(U)((U)x - (U)y)$ 计算。换言之，运算符计算 x 和 y 的有序值之间的差，结果类型是枚举的基础类型。

```
E operator -(E x, U y);
```

该运算符严格按 $(E)((U)x - y)$ 计算。换言之，该运算符从枚举的基础类型中减去一个值，得到枚举的值。

- 委托移除。每个委托类型都隐式提供以下预定义运算符，其中 D 是委托类型：

```
D operator -(D x, D y);
```

当两个操作数均为某个委托类型 D 时，二元 $-$ 运算符执行委托移除。如果操作数具有不同的委托类型，则发生编译时错误。如果第一个操作数为 `null`，则运算结果为 `null`。否则，如果第二个操作数为 `null`，则运算结果为第一个操作数的值。否则，两个操作数都表示包含一项或多项的调用列表 (§ 15.1)，并且只要第二个操作数列表是第一个操作数列表的适当的邻接子列表，那么结果就是从第一个操作数的调用列表中移除了第二个操作数的调用列表所含各项后的一个新调用列表（为确定子列表是否相等，用委托相等运算符比较相对应的项，请参见 § 7.9.8）。否则，结果为左操作数的值。在此过程中两个操作数的列表均未被更改。如果第二个操作数的列表与第一个操作数的列表中的多个邻接项子列表相匹配，则移除最右边的那个匹配邻接项的子列表。如果移除导致空列表，则结果为 `null`。例如：

```
delegate void D(int x);
class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd1; // => M1 + M2 + M2
        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2; // => M2 + M1
        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2; // => M1 + M1
        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1; // => M1 + M2
        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1; // => M1 + M2 + M2 + M1
    }
}
```

7.8 移位运算符

`<<` 和 `>>` 运算符用于执行移位运算。

shift-expression: (移位表达式:)

additive-expression (加法表达式)

shift-expression << additive-expression (移位表达式 << 加法表达式)

shift-expression >> additive-expression (移位表达式 >> 加法表达式)

对于 $x \ll \text{count}$ 或 $x \gg \text{count}$ 形式的运算, 应用二元运算符重载决策 (§ 7.2.4) 以选择特定的运算符实现。操作数转换为所选运算符的参数类型, 结果的类型是该运算符的返回类型。

当声明重载移位运算符时, 第一个操作数的类型必须总是包含运算符声明的类或结构, 并且第二个操作数的类型必须总是 `int`。

下面列出了预定义的移位运算符。

- 左移位:

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

\ll 运算符将 x 向左位移若干个位, 具体计算方法如下所述。

放弃 x 中经移位后会超出结果类型范围的那些高序位, 将其余的位向左位移, 将空出来的低序位均设置为零。

- 右移位:

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

\gg 运算符将 x 向右位移若干个位, 具体计算方法如下所述。

当 x 为 `int` 或 `long` 类型时, 放弃 x 的低序位, 将剩余的位向右位移, 如果 x 非负, 则将高序空位位置设置为零, 如果 x 为负, 则将其设置为 1。

当 x 为 `uint` 或 `ulong` 类型时, 放弃 x 的低序位, 将剩余的位向右位移, 并将高序空位位置设置为零。

对于预定义运算符, 位移的位数按下面这样计算:

- 当 x 的类型为 `int` 或 `uint` 时, 位移计数由 `count` 的低序的 5 位给出。换言之, 位移计数由 `count & 0x1F` 计算出。
- 当 x 的类型为 `long` 或 `ulong` 时, 位移计数由 `count` 的低序的 6 位给出。换言之, 位移计数由 `count & 0x3F` 计算出。

如果计算位移计数的结果为零, 则移位运算符只返回 x 的值。

移位运算从不会导致溢出, 并且在 `checked` 和 `unchecked` 上下文中产生的结果相同。

当 \gg 运算符的左操作数为有符号的整型时, 该运算符执行算术右移位, 在此过程中, 操作数的最有效位 (符号位) 的值扩展到高序空位位置。当 \gg 运算符的左操作数为无符号的整型时, 该运算符执行逻辑右移位, 在此过程中, 高序空位位置总是设置为零。若要执行与由操作数类型确定的不同的移位运算, 可以使用显式强制转换。例如, 如果 x 是 `int` 类型的变量, 则 `unchecked((int)((uint)x >> y))` 运算执行 x 的逻辑右移位。

7.9 关系和类型测试运算符

==, !=, <, >, <=, >=, is 和 as 运算符称为关系和类型测试运算符。

relational-expression: (关系表达式:)

shift-expression (移位表达式)

relational-expression < shift-expression (关系表达式 < 移位表达式)

relational-expression > shift-expression (关系表达式 > 移位表达式)

relational-expression <= shift-expression (关系表达式 <= 移位表达式)

relational-expression >= shift-expression (关系表达式 >= 移位表达式)

relational-expression is type (关系表达式 is 类型)

relational-expression as type (关系表达式 as 类型)

equality-expression: (相等表达式:)

relational-expression (关系表达式)

equality-expression == relational-expression (相等表达式 == 关系表达式)

equality-expression != relational-expression (相等表达式 != 关系表达式)

is 运算符的介绍详见 § 7.9.9, as 运算符的介绍详见 § 7.9.10。

==, !=, <, >, <= 和 >= 运算符是比较运算符。对于形式为 x op y (其中 op 为比较运算符) 的运算, 应用重载决策 (§ 7.2.4) 以选择特定的运算符实现。操作数转换为所选运算符的参数类型, 结果的类型是该运算符的返回类型。

预定义的比较运算符详见下面各节的介绍。所有预定义的比较运算符都返回 bool 类型的结果, 详见表 7.4。

表 7.4 预定义的比较运算符的返回结果

操 作	结 果
x==y	如果 x 等于 y, 则为 true, 否则为 false
x!=y	如果 x 不等于 y, 则为 true, 否则为 false
x<y	如果 x 小于 y, 则为 true, 否则为 false
x>y	如果 x 大于 y, 则为 true, 否则为 false
x<=y	如果 x 小于等于 y, 则为 true, 否则为 false
x>=y	如果 x 大于等于 y, 则为 true, 否则为 false

7.9.1 整数比较运算符

预定义的整数比较运算符为:

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
```



```

bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);

```

这些运算符比较两个整数操作数的数值并返回一个 `bool` 值，该值指示特定的关系是 `true` 还是 `false`。

7.9.2 浮点比较运算符

预定义的浮点比较运算符为：

```

bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);

```

这些运算符根据 IEEE 754 标准法则比较操作数。

- 如果两个操作数中的任何一个为 NaN，则对于除 `!=`（对于此运算符，结果为 `true`）外的所有运算符，结果为 `false`。对于任何两个操作数，`x != y` 总是产生与 `!(x == y)` 相同的结果。然而，当一个操作数或两个操作数为 NaN 时，`<`，`>`，`<=` 和 `>=` 运算符不产生与其对应的反向运算符的逻辑否定相同的结果。例如，如果 `x` 和 `y` 中的任何一个为 NaN，则 `x < y` 为 `false`，而 `!(x >= y)` 为 `true`。
- 当两个操作数都不为 NaN 时，这些运算符就按下列排序来比较两个浮点操作数

的值 $-\infty < -\max < \dots < -\min < -0.0 == +0.0 < +\min < \dots < +\max < +\infty$ 这里的 \min 和 \max 是可以用给定浮点格式表示的最小和最大正有限值。这样排序的显著特点是：

- 负零和正零被视为相等。
- 负无穷大被视为小于所有其他值，但等于其他负无穷大。
- 正无穷大被视为大于所有其他值，但等于其他正无穷大。

7.9.3 小数比较运算符

预定义的小数比较运算符为：

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

这些运算符中每一个都比较两个小数类型的操作数的数值并返回一个 `bool` 值，该值指示特定的关系是 `true` 还是 `false`。各小数比较等效于使用 `System.Decimal` 类型的相应关系运算符或相等运算符。

7.9.4 布尔相等运算符

预定义的布尔相等运算符为：

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

如果 x 和 y 都为 `true`，或者如果 x 和 y 都为 `false`，则 `==` 的结果为 `true`。否则，结果为 `false`。

如果 x 和 y 都为 `true`，或者 x 和 y 都为 `false`，则 `!=` 的结果为 `false`。否则，结果为 `true`。当操作数为 `bool` 类型时，`!=` 运算符产生与 `^` 运算符相同的结果。

7.9.5 枚举比较运算符

每种枚举类型都隐式提供下列预定义的比较运算符：

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

$x \text{ op } y$ (其中 x 和 y 是具有基础类型 U 的枚举类型 E 的表达式， op 是比较运算符之一) 的计算结果与 $((U)x) \text{ op } ((U)y)$ 的计算结果完全相同。换言之，枚举类型比较运算

符只比较两个操作数的基础整数值。

7.9.6 引用类型相等运算符

预定义的引用类型相等运算符为：

```
bool operator ==(object x, object y);  
bool operator !=(object x, object y);
```

这些运算符返回两个引用是相等还是不相等的比较结果。

由于预定义的引用类型相等运算符接受 `object` 类型的操作数，因此它们适用于所有那些没有为自己声明适用的 `operator ==` 和 `operator !=` 成员的类型。相反，任何适用的用户定义的相等运算符都有效地隐藏上述预定义的引用类型相等运算符。

预定义引用类型相等运算符要求操作数是引用类型值或 `null` 值；此外，它们还要求存在从一种操作数类型到另一种操作数类型的标准隐式转换（§ 6.3.1）。除非这两个条件都为真，否则将发生编译时错误。这些规则中值得注意的含义是：

- 使用预定义的引用类型相等运算符比较两个在编译时已能确定是不相同的引用时，会导致编译时错误。例如，如果操作数的编译时类型分属两个类类型 `A` 和 `B`，并且 `A` 和 `B` 都不从对方派生，则两个操作数不可能引用同一对象。因此，此运算被认为是编译时错误。
- 预定义的引用类型相等运算符不允许比较值类型操作数。因此，除非结构类型声明自己的相等运算符，否则不可能比较该结构类型的值。
- 预定义的引用类型相等运算符从不会导致对它们的操作数执行装箱操作。执行此类装箱操作毫无意义，这是因为对新分配的已装箱实例的引用必将不同于所有其他引用。

对于 `x == y` 或 `x != y` 形式的运算，如果存在任何适用的 `operator ==` 或 `operator !=`，运算符重载决策（§ 7.2.4）规则将选择该运算符而不是上述的预定义的引用类型相等运算符。不过，始终可以通过将一个或两个操作数显式强制转换为 `object` 类型来选择预定义的引用类型相等运算符。示例

```
using System;  
class Test  
{  
    static void Main() {  
        string s = "Test";  
        string t = string.Copy(s);  
        Console.WriteLine(s == t);  
        Console.WriteLine((object)s == t);  
        Console.WriteLine(s == (object)t);  
        Console.WriteLine((object)s == (object)t);  
    }  
}
```

产生输出

```
True  
False
```

```
False
False
```

变量 `s` 和 `t` 引用两个包含相同字符的不同 `string` 实例。第一个比较输出 `True`，这是因为是在两个操作数都为 `string` 类型时选定预定义的字符串相等运算符 (§ 7.9.7)。其余的比较全都输出 `False`，这是因为是在一个或两个操作数为 `object` 类型时选定预定义的引用类型相等运算符。

注意，以上技术对值类型没有意义。示例

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

输出 `False`，这是因为强制转换创建对已装箱 `int` 值的两个单独实例的引用。

7.9.7 字符串相等运算符

预定义的字符串相等运算符为：

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

当下列条件中有一个为真时，两个 `string` 值被视为相等：

两个值都为 `null`。

两个值都是对字符串实例的非空引用，这两个字符串不仅具有相同的长度，而且在每个字符位置上的字符也都彼此相同。

字符串相等运算符比较的是字符串的值而不是对字符串的引用。当两个单独的字符串实例包含完全相同的字符序列时，字符串的值相等，但引用不相同。正如 § 7.9.6 中所描述的那样，引用类型相等运算符可用于比较字符串引用而不是字符串值。

7.9.8 委托相等运算符

每个委托类型都隐式地提供下列预定义的比较运算符：

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

两个下面这样的委托实例被视为相等。

- 如果两个委托实例中有一个为 `null`，则当且仅当它们都为 `null` 时相等。
- 如果两个委托实例中有一个具有包含一个项的调用列表 (§ 15.1)，则当且仅当另一个委托实例也具有包含一个项的调用列表且符合下列条件之一时，两个委托实例相等：

- 两者引用同一静态方法;
- 两者都引用同一目标对象的同一非静态方法。
- 如果两个委托实例中有一个具有包含两项或更多项的调用列表,则当且仅当它们的调用列表长度相同,并且一个实例的调用列表中的每项依次等于另一个的调用列表中的相应项时,这两个委托实例相等。

注意,根据上面的定义,只要委托具有相同的返回类型和参数类型,即使它们的类型不同也被视为相等。

7.9.9 is 运算符

is 运算符用于动态检查对象的运行时类型是否与给定类型兼容。`e is T` 运算(其中 `e` 为表达式, `T` 为类型)的结果是布尔值,表示 `e` 的类型是否可通过引用转换、装箱转换或取消装箱转换成功地转换为类型 `T`。

- 如果 `e` 的编译时类型与 `T` 相同,或存在从 `e` 的编译时类型到 `T` 的隐式引用转换 (§ 6.1.4) 或装箱转换 (§ 6.1.5), 那么:
 - 如果 `e` 为引用类型,则运算结果等效于计算 `e != null`。
 - 如果 `e` 为值类型,则运算结果为 `true`。
- 否则,如果存在从 `e` 的编译时类型到 `T` 的显式引用转换 (§ 6.2.3) 或取消装箱转换 (§ 6.2.4), 则执行动态类型检查:
 - 如果 `e` 的值为 `null`, 则结果为 `false`。
 - 否则,假设 `R` 为 `e` 所引用的实例的运行时类型。如果 `R` 和 `T` 的类型相同,或者如果 `R` 为引用类型且存在从 `R` 到 `T` 的隐式引用转换,或者如果 `R` 为值类型且 `T` 为 `R` 实现的接口类型,则结果为 `true`。
 - 否则,结果为 `false`。
- 否则,不可能实现从 `e` 到类型 `T` 的引用转换或装箱转换,且运算结果为 `false`。

注意, `is` 运算符只考虑引用转换、装箱转换和取消装箱转换。其他转换(如用户定义的转换)不在 `is` 运算符考虑之列。

7.9.10 as 运算符

as 运算符用于将一个值显式地转换(使用引用转换或装箱转换)为一个给定的引用类型。与强制转换表达式 (§ 7.6.6) 不同,as 运算符从不引发异常。它采用的是:如果指定的转换不可能实施,则运算结果为 `null`。

在 `e as T` 形式的运算中,`e` 必须是表达式,`T` 必须是引用类型。该运算的结果属于类型 `T`,且总是可归类为值类别。运算按下面这样计算:

- 如果 `e` 的编译时类型与 `T` 相同,则结果就是 `e` 的值。
- 如果存在从 `e` 的编译时类型到 `T` 的隐式引用转换 (§ 6.1.4) 或装箱转换 (§ 6.1.5), 则执行该转换,且该转换的结果就是运算结果。
- 如果存在从 `e` 的编译时类型到 `T` 的显式引用转换 (§ 6.2.3), 则执行动态类型

检查。

- 如果 *e* 的值为 `null`，则结果为具有编译时类型 *T* 的值 `null`。
- 否则，假设 *R* 为 *e* 引用的实例的运行时类型。如果 *R* 和 *T* 的类型相同，或者如果 *R* 为引用类型且存在从 *R* 到 *T* 的隐式引用转换，或者如果 *R* 为值类型且 *T* 是由 *R* 实现的一个接口类型，则结果为由 *e* 给定的具有编译时类型 *T* 的引用。
- 否则，结果为具有编译时类型 *T* 的值 `null`。
- 否则，指定的转换根本不可能实现，且发生编译时错误。

注意，`as` 运算符只执行引用转换和装箱转换。不可能使用 `as` 运算符执行其他转换（如用户定义的转换），应改为使用强制转换表达式来执行这些转换。

7.10 逻辑运算符

`&`，`^` 和 `|` 运算符称为逻辑运算符。

`and-expression`: (与表达式:)

`equality-expression` (相等表达式)

`and-expression & equality-expression` (与表达式 `&` 相等表达式)

`exclusive-or-expression`: (异或表达式:)

`and-expression` (与表达式)

`exclusive-or-expression ^ and-expression` (异或表达式 `^` 与表达式)

`inclusive-or-expression`: (或表达式:)

`exclusive-or-expression` (异或表达式)

`inclusive-or-expression | exclusive-or-expression` (异或表达式 `|` 异或表达式)

对于 `x op y` 形式的运算（其中 `op` 为逻辑运算符），应用重载决策（§ 7.2.4）以选择一个特定的运算符实现。操作数转换为所选运算符的参数类型，结果的类型是该运算符的返回类型。

下面的章节介绍了预定义的逻辑运算符。

7.10.1 整数逻辑运算符

预定义的整数逻辑运算符为：

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);
```

```
int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

& 运算符计算两个操作数的按位逻辑 AND，| 运算符计算两个操作数的按位逻辑 OR，而运算符 ^ 计算两个操作数的按位逻辑 OR。这些运算不会产生溢出。

7.10.2 枚举逻辑运算符

每个枚举类型 E 都隐式地提供下列预定义的逻辑运算符：

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

$x \text{ op } y$ (其中 x 和 y 是具有基础类型 U 的枚举类型 E 的表达式, op 是逻辑运算符) 的计算结果与 $(E)((U)x \text{ op } (U)y)$ 的计算结果完全相同。换言之, 枚举类型逻辑运算符直接对两个操作数的基础类型执行逻辑运算。

7.10.3 布尔逻辑运算符

预定义的布尔逻辑运算符为：

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

如果 x 和 y 均为 `true`, 则 $x \& y$ 的结果为 `true`。否则, 结果为 `false`。

如果 x 或 y 为 `true`, 则 $x|y$ 的结果为 `true`。否则, 结果为 `false`。

如果 x 为 `true` 而 y 为 `false`, 或者 x 为 `false` 而 y 为 `true`, 则 x^y 的结果为 `true`。否则, 结果为 `false`。当操作数为 `bool` 类型时, \wedge 运算符计算结果与 `!=` 运算符相同。

7.11 条件逻辑运算符

运算符 `&&` 和 `||` 是条件逻辑运算符, 也称为“短路”逻辑运算符。

conditional-and-expression: (条件与表达式:)

inclusive-or-expression (或表达式)

conditional-and-expression `&&` **inclusive-or-expression** (条件与表达式 `&&` 或表达式)

conditional-or-expression: (条件或表达式:)

conditional-and-expression (条件与表达式)

conditional-or-expression `||` **conditional-and-expression** (条件或表达式 `||` 条件与表达式)

&& 和 || 运算符是 & 和 | 运算符的条件版本:

- $x \&\& y$ 运算对应于 $x \& y$ 运算, 但仅当 x 为 `true` 时才计算 y 。
- $x \parallel y$ 运算对应于 $x | y$ 运算, 但仅当 x 为 `false` 时才计算 y 。

$x \&\& y$ 或 $x \parallel y$ 形式的运算通过采用重载决策 (§ 7.2.4) 按运算被写为 $x \& y$ 或 $x | y$ 来处理。

- 如果重载决策未能找到单个最佳运算符, 或者重载决策选择一个预定义的整数逻辑运算符, 则发生编译时错误。
- 如果选定的运算符是一个预定义的布尔逻辑运算符 (§ 7.10.2), 则运算按 § 7.11.1 中所描述的那样进行处理。
- 否则, 选定的运算符为用户定义的运算符, 且运算按 § 7.11.2 中所描述的那样进行处理。

不可能直接重载条件逻辑运算符。不过, 由于条件逻辑运算符按通常的逻辑运算符计算, 因此在某些限制条件下, 通常的逻辑运算符的重载也被视为条件逻辑运算符的重载。§ 7.11.2 对此有进一步说明。

7.11.1 布尔条件逻辑运算符

如果 && 或 || 的操作数为 `bool` 类型时, 或者如果操作数的类型本身未定义适用的 `operator &` 或 `operator |`, 但确实定义了到 `bool` 的隐式转换, 则运算按下面这样处理。

- 运算 $x \&\& y$ 的求值过程相当于 $x ? y : \text{false}$ 。换言之, 首先计算 x 并将其转换为 `bool` 类型。如果 x 为 `true`, 则计算 y 并将其转换为 `bool` 类型, 并且这成为运算结果。否则, 运算结果为 `false`。
- 运算 $x \parallel y$ 的求值过程相当于 $x ? \text{true} : y$ 。换言之, 首先计算 x 并将其转换为 `bool` 类型。如果 x 为 `true`, 则运算结果为 `true`。否则, 计算 y 并将其转换为 `bool` 类型, 并把这作为运算结果。

7.11.2 用户定义的条件逻辑运算符

当 && 或 || 的操作数所属的类型声明了适用的用户定义的 `operator &` 或 `operator |` 时, 下列两个条件必须都为真 (其中 T 是声明的选定运算符的类型)。

- 选定运算符的返回类型和每个参数的类型都必须为 T 。换言之, 该运算符必须计算类型为 T 的两个操作数的逻辑 AND 或逻辑 OR, 且必须返回类型为 T 的结果。
- T 必须包含关于 `operator true` 和 `operator false` 的声明。

如果这两个要求中有一个未满足, 则发生编译时错误。如果这两个要求都满足, 则通过将用户定义的 `operator true` 或 `operator false` 与选定的用户定义的运算符组合在一起来计算 && 运算或 || 运算。

- $x \&\& y$ 运算按 $T.\text{false}(x) ? x : T.\&(x, y)$ 进行计算, 其中 $T.\text{false}(x)$ 是 T 中声明的 `operator false` 的调用, $T.\&(x, y)$ 是选定 `operator &` 的调用。换言之, 首先计

算 x ，并对结果调用 `operator false` 以确定 x 是否肯定为假。如果 x 肯定为假，则运算结果为先前为 x 计算的值。否则将计算 y ，并对先前为 x 计算的值和为 y 计算的值调用选定的 `operator &` 以产生运算结果。

- $x \parallel y$ 运算按 `T.true(x) ? x : T.l(x, y)` 进行计算，其中 `T.true(x)` 是 `T` 中声明的 `operator true` 的调用，`T.l(x, y)` 是选定的 `operator l` 的调用。换言之，首先计算 x ，并对结果调用 `operator true` 以确定 x 是否肯定为真。然后，如果 x 肯定为真，则运算结果为先前为 x 计算的值。否则将计算 y ，并对先前为 x 计算的值和为 y 计算的值调用选定的 `operator l` 以产生运算结果。

在这两个运算中，对于 x 给定的表达式只计算一次，对于 y 给定的表达式要么不计算，要么只计算一次。

有关实现 `operator true` 和 `operator false` 的类型的示例，请参见 § 11.4.2。

7.12 条件运算符

?: 运算符称为条件运算符。有时，它也称为三元运算符。

`conditional-expression`: (条件表达式:)

`conditional-or-expression` (条件或表达式)

`conditional-or-expression ? expression : expression` (条件或表达式 ? 表达式 : 表达式)

$b ? x : y$ 形式的条件表达式首先计算条件 b 。如果 b 为 `true`，则计算 x ，并且它作为运算结果。否则计算 y ，并且它作为运算结果。条件表达式从不同时计算 x 和 y 。

条件运算符向右关联，表示运算从右到左分组。例如， $a ? b : c ? d : e$ 形式的表达式按 $a ? b : (c ? d : e)$ 计算。

?: 运算符的第一个操作数必须是可以隐式转换为 `bool` 的类型的表达式，或者是可以实现 `operator true` 的类型的表达式。如果两个要求都不满足，则发生编译时错误。

?: 运算符的第二个和第三个操作数决定了条件表达式的类型。设 X 和 Y 为第二个和第三个操作数所属的类型。

- 如果 X 和 Y 的类型相同，则此类型为该条件表达式的类型。
- 如果存在从 X 到 Y 的隐式转换 (§ 6.1)，但不存在从 Y 到 X 的隐式转换，则 Y 为条件表达式的类型。
- 如果存在从 Y 到 X 的隐式转换，但不存在从 X 到 Y 的隐式转换，则 X 为条件表达式的类型。
- 否则，无法确定条件表达式的类型，且发生编译时错误。

$b ? x : y$ 形式的条件表达式的运行时处理包括以下步骤。

- 首先计算 b ，并确定 b 的 `bool` 值：
 - 如果存在从 b 的类型到 `bool` 的隐式转换，则执行该隐式转换以产生 `bool` 值。
 - 否则，调用 b 的类型中定义的 `operator true` 以产生 `bool` 值。
- 如果以上步骤产生的 `bool` 值为 `true`，则计算 x 并将其转换为条件表达式的类

型，且这成为条件表达式的结果。

- 否则，计算 y 并将其转换为条件表达式的类型，且这成为条件表达式的结果。

7.13 赋值运算符

赋值运算符为变量、属性、事件或索引器元素赋新值。

assignment: (赋值:)

unary-expression assignment-operator expression (一元表达式 赋值运算符 表达式)

assignment-operator: one of (赋值运算符: 下列之一)

= += -= *= /= %= &= |= ^= <<= >>=

赋值的左操作数必须是属于变量、属性访问、索引器访问或事件访问类别的表达式。

= 运算符称为**简单赋值运算符** (simple assignment operator)。它将右操作数的值赋予左操作数给定的变量、属性或索引器元素。简单赋值运算符的左操作数一般不可能是一个事件访问 (§ 10.7.1 中描述的例外)。简单赋值运算符的介绍详见 § 7.13.1。

除 = 运算符以外的赋值运算符称为**复合赋值运算符** (compound assignment operator)。这些运算符对两个操作数执行指示的运算，然后将结果值赋予左操作数给定的变量、属性或索引器元素。复合赋值运算符的介绍详见 § 7.13.2。

以事件访问表达式作为左操作数的 += 和 -= 运算符称为事件赋值运算符。当左操作数是一个事件访问时，其他赋值运算符都是无效的。对事件赋值运算符将在 § 7.13.3 中进行介绍。

赋值运算符为向右关联，即此类运算从右到左分组。例如， $a = b = c$ 形式的表达式按 $a = (b = c)$ 计算。

7.13.1 简单赋值

在简单赋值中，右操作数表达式所属的类型必须可隐式地转换为左操作数所属的类型。运算将右操作数的值赋予左操作数指定的变量、属性或索引器元素。

简单赋值表达式的结果是赋予左操作数的值。结果的类型与左操作数相同，且始终为值类别。

如果左操作数为属性或索引器访问，则该属性或索引器必须具有 set 访问器。否则，将发生编译时错误。

$x = y$ 形式的简单赋值的运行时处理包括以下步骤。

- 如果 x 属于变量，则：
 - 计算 x 以产生变量。
 - 如果要求，则计算 y ，必要时还需通过隐式转换 (§ 6.1) 将其转换为 x 的类型。
 - 如果 x 给定的变量是引用类型的数组元素，则执行运行时检查以确保为 y 计算的值与以 x 为其元素的那个数组实例兼容。如果 y 为 null，或存在从 y 引用

的实例的实际类型到包含 x 的数组实例的实际元素类型的隐式引用转换 (§ 6.1.4), 则检查成功。否则, 引发 `System.ArrayTypeMismatchException`。

- y 的计算和转换后所产生的值存储在 x 的计算所确定的位置中。

● 如果 x 属于属性或索引器访问, 则:

- 计算与 x 关联的实例表达式 (如果 x 不是 `static`) 和参数列表 (如果 x 是索引器访问), 结果用于后面的对 `set` 访问器调用。

- 计算 y , 必要时还需通过隐式转换 (§ 6.1) 将其转换为 x 的类型。

- 调用 x 的 `set` 访问器, 并将 y 的上述结果值作为该访问器的 `value` 参数。

如果存在从 B 到 A 的隐式引用转换, 则数组协方差规则 (§ 12.5) 允许数组类型 $A[]$ 的值成为对数组类型 $B[]$ 的实例的引用。由于这些规则, 对引用类型的数组元素的赋值需要运行时检查, 以确保所赋的值与数组实例兼容。在下面的示例中, 最后的赋值导致引发 `System.ArrayTypeMismatchException`, 这是因为 `ArrayList` 的实例不能存储在 `string[]` 的元素中。

```
string[] sa = new string[10];
object[] oa = sa;
oa[0] = null;           // Ok
oa[1] = "Hello";        // Ok
oa[2] = new ArrayList(); // ArrayTypeMismatchException
```

当结构类型中声明的属性或索引器是赋值的目标时, 与属性或索引器访问关联的实例表达式必须为变量类别。如果该实例表达式归类为值类别, 则发生编译时错误。由于 § 7.5.4 中所说明的原因, 同样的规则也适用于字段。

给定下列声明:

```
struct Point
{
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int X {
        get { return x; }
        set { x = value; }
    }
    public int Y {
        get { return y; }
        set { y = value; }
    }
}
struct Rectangle
{
    Point a, b;
    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }
    public Point A {
        get { return a; }
        set { a = value; }
    }
}
```

```

public Point B {
    get { return b; }
    set { b = value; }
}

```

在下面的示例中，由于 `p` 和 `r` 为变量，因此允许对 `p.X`、`p.Y`、`r.A` 和 `r.B` 赋值。

```

Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;

```

但是，在以下示例中，由于 `r.A` 和 `r.B` 不是变量，所以赋值全部无效。

```

Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;

```

7.13.2 复合赋值

`x op= y` 形式的运算是这样来处理的：将二元运算符重载决策 (§ 7.2.4) 应用于运算 `x op y`。

- 如果选定的运算符的返回类型可隐式转换为 `x` 的类型，则运算按 `x = x op y` 计算，但 `x` 只计算一次。
- 如果选定运算符是预定义的运算符，选定运算符的返回类型可显式转换为 `x` 的类型，并且 `y` 可隐式转换为 `x` 的类型，则运算按 `x = (T)(x op y)` 计算（其中 `T` 是 `x` 的类型），但 `x` 只计算一次。
- 否则，复合赋值无效，且发生编译时错误。

术语“只计算一次”表示：在 `x op y` 的计算中，任何 `x` 的要素表达式的计算结果都临时保存起来，然后在执行对 `x` 的赋值时重用这些结果。例如，在计算赋值 `A()[B()] += C()` 时（其中 `A` 为返回 `int[]` 的方法，`B` 和 `C` 为返回 `int` 的方法），按 `A`，`B`，`C` 的顺序只调用这些方法一次。

当复合赋值的左操作数为属性访问或索引器访问时，属性或索引器必须同时具有 `get` 访问器和 `set` 访问器。否则，将发生编译时错误。

上面的第二条规则允许在某些上下文中将 `x op= y` 按 `x = (T)(x op y)` 计算。当左操作数为 `sbyte`，`byte`，`short`，`ushort` 或 `char` 类型时，按此规则，预定义的运算符可用来构造复合运算符。甚至当两个参数都为这些类型之一时，预定义的运算符也产生 `int` 类型的结果，详见 § 7.2.6.2 中的介绍。因此，不进行强制转换，就不可能把结果赋值给左操作数。

此规则对预定义运算符的直观效果只是：如果同时允许 `x op y` 和 `x = y`，则允许 `x op= y`。在下面的示例中，每个错误的直观理由是对应的简单赋值也发生错误。

```

byte b = 0;
char ch = '\0';

```

```

int i = 0;
b += 1;           // Ok
b += 1000;        // 错误, 不允许 b = 1000
b += i;           // 错误, 不允许 b = i
b += (byte)i;     // Ok
ch += 1;          // 错误, 不允许 ch = 1
ch += (char)1;    // Ok

```

7.13.3 事件赋值

如果 `+=` 或 `-=` 运算符的左操作数属于事件访问类别, 则表达式按下面这样计算。

- 计算事件访问的实例表达式 (如果有)。
- 计算 `+=` 或 `-=` 运算符的右操作数, 如果需要, 通过隐式转换 (§ 6.1) 转换为左操作数的类型。
- 调用该事件的事件访问器, 所需的参数列表由右操作数 (经过计算和必要的转换后) 组成。如果运算符是 `+=`, 则调用 `add` 访问器。如果运算符是 `-=`, 则调用 `remove` 访问器。

事件赋值表达式不产生值。因此, 事件赋值表达式只在语句表达式 (§ 8.6) 的上下文中有效。

7.14 表达式

表达式可以是条件表达式或赋值。

expression: (表达式:)

conditional-expression (条件表达式)

assignment (赋值)

7.15 常数表达式

常数表达式是在编译时可以完全计算出结果的表达式。

constant-expression: (常数表达式:)

expression (表达式)

常数表达式的类型可以为下列之一: `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal`、`bool`、`string`、任何枚举类型或空类型。常数表达式中允许下列构造。

- 文本 (包括 `null`)。
- 对类和结构类型的 `const` 成员的引用。
- 对枚举类型的成员的引用。
- 带括号的子表达式, 其自身是常量表达式。
- 强制转换表达式 (前提是目标类型为以上列出的类型之一)。

- 预定义的一元运算符`+`, `-`, `!` 和 `~`。
- 预定义的二元运算符 `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=` 和 `>=` (前提是每个操作数都为上面列出的类型)。
- 条件运算符`?:`。

只要表达式属于上面列出的类型之一且只包含上面列出的构造, 就在编译时计算该表达式。即使该表达式是另一个包含有非常数构造的较大表达式的子表达式, 也是如此。

常数表达式的编译时计算使用与非常数表达式的运行时计算相同的规则, 区别仅在于: 当出现错误时, 运行时计算引发异常, 而编译时计算导致编译时错误。

除非常数表达式被显式放置在 `unchecked` 上下文中, 否则在表达式的编译时计算期间, 整型算术运算和转换中发生的溢出总是导致编译时错误 (§ 7.5.12)。

常数表达式出现在下面列出的上下文中。在这些上下文中, 如果无法在编译时充分计算表达式, 则导致编译时错误。

- 常数声明 (§ 10.3)。
- 枚举成员声明 (§ 14.3)。
- `switch` 语句的 `case` 标签 (§ 8.7.2)。
- `goto case` 语句 (§ 8.9.3)。
- 包含初始值设定项的数组创建表达式 (§ 7.5.10.2) 中的维度长度。
- 特性 (§ 17)。

只要常数表达式的值在目标类型的范围内, 隐式常数表达式转换 (§ 6.1.6) 就允许将 `int` 类型的常数表达式转换为 `sbyte`, `byte`, `short`, `ushort`, `uint` 或 `ulong`。

7.16 布尔表达式

布尔表达式是产生 `bool` 类型结果的表达式。

boolean-expression: (布尔表达式:)

expression (表达式)

`if` 语句 (§ 8.7.1)、`while` 语句 (§ 8.8.1)、`do` 语句 (§ 8.8.2) 或 `for` 语句 (§ 8.8.3) 的控制条件表达式都是布尔表达式。`?:` 运算符 (§ 7.12) 的控制条件表达式遵守与布尔表达式相同的规则, 但由于运算符优先级的缘故, 被归为“条件或表达式”。

要求布尔表达式的类型或者可隐式地转换为 `bool` 的类型, 或者实现了 `operator true` 的类型。如果两个要求都不满足, 则发生编译时错误。

当布尔表达式的类型不能隐式转换为 `bool` 但它的确实现了 `operator true` 时, 则在完成表达式计算后, 会调用该类型提供的 `operator true` 以产生 `bool` 值。

§ 11.4.2 中的 `DBBool` 结构类型提供了一个实现了 `operator true` 和 `operator false` 的类型的示例。

第 8 章 语句

C# 提供了各种不同的语句。使用 C 和 C++ 编过程序的开发人员对这些语句中的大多数都会非常熟悉。

statement: (语句:)

labeled-statement (标记语句)

declaration-statement (声明语句)

embedded-statement (嵌入语句)

embedded-statement: (嵌入语句:)

block (块)

empty-statement (空语句)

expression-statement (表达式语句)

selection-statement (选择语句)

iteration-statement (迭代语句)

jump-statement (跳转语句)

try-statement (try 语句)

checked-statement (checked 语句)

unchecked-statement (unchecked 语句)

lock-statement (lock 语句)

using-statement (using 语句)

非结束性嵌入语句用于在其他语句内出现的语句。在使用一般语句的地方使用嵌入语句，排除在这些上下文中使用声明语句和标记语句的可能。示例

```
void F(bool b) {  
    if (b)  
        int i = 44;  
}
```

将导致编译时错误，原因是 if 语句的 if 分支要求“嵌入语句”而不是“语句”。上述代码的效果是：声明了变量 *i*，却永远无法使用它。但是请注意，如果是将 *i* 的声明放置在一个块中，则该示例就是有效的。

8.1 结束点和可达性

每个语句都有一个结束点。直观地讲，语句的结束点是紧跟在语句后面的那个位置。复合语句（包含嵌入语句的语句）的执行规则规定了当控制到达一个嵌入语句的结束点时所采取的操作。例如，当控制到达块中某个语句的结束点时，控制就转移到该块中的下一

个语句。

如果执行流程可能到达某个语句，则称该语句是可到达的。相反，如果某个语句不可能被执行，则称该语句是不可到达的。

在下面的示例中第二个 `Console.WriteLine` 调用是不可到达的，这是因为不可能执行该语句。

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
    Label:
    Console.WriteLine("reachable");
}
```

如果编译器确定某个语句是不可到达的，就会报出警告。准确地说，语句不可到达不算是错误。

为了确定某个特定的语句或结束点是否可到达，编译器根据为各语句定义的可到达性规则进行控制流分析。控制流分析会考虑那些能控制语句行为的常数表达式（§ 7.15）的值，但不考虑非常数表达式的可能值。换句话说，出于控制流分析的目的，给定类型的非常数表达式被认为具有该类型的任何可能值。

请看下面的示例：

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

其中，`if` 语句的布尔表达式是常数表达式，原因是 `==` 运算符的两个操作数都是常数。由于该常数表达式在编译时进行计算并产生值 `false`，所以 `Console.WriteLine` 调用被认为是不可到达的。但是，如果 `i` 更改为局部变量，则 `Console.WriteLine` 调用被认为是可到达的，即使它实际上永远不会被执行。

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

函数成员的块始终被认为是可到达的。通过依次计算块中各语句的可到达性规则，可以确定任何给定语句的可到达性。

请看下面的示例：

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

其中，第二个 `Console.WriteLine` 的可到达性按下面的规则确定。

- 第一个 `Console.WriteLine` 表达式语句是可到达的，原因是 `F` 方法的块是可到达的。
- 第一个 `Console.WriteLine` 表达式语句的结束点是可到达的，原因是该语句是可

到达的。

- if 语句是可到达的，原因是第一个 `Console.WriteLine` 表达式语句的结束点是可到达的。
- 第二个 `Console.WriteLine` 表达式语句是可到达的，原因是 if 语句的布尔表达式不是常数值 `false`。

在下列两种情况下，如果某个语句的结束点是可以到达的，则会出现编译时错误：

- 由于 `switch` 语句不允许一个 `switch` 节“贯穿”到下一个 `switch` 节，因此如果一个 `switch` 节的语句列表的结束点是可到达的，则会出现编译时错误。如果发生此错误，那么它通常表明该处遗漏了一个 `break` 语句。
- 如果计算某个值的函数成员的块的结束点是可到达的，将会出现编译时错误。如果发生此错误，那么它通常表明该处遗漏了一个 `return` 语句。

8.2 块

块用于在只能使用单个语句的上下文中编写多个语句。

block: (块:)

{ statement-list_{opt} } ({ 语句列表_{可选} })

块由一个括在大括号内的可选语句列表组成。如果没有此语句列表，则称块是空的。

块可以包含声明语句 (§ 8.5)。在块中声明的局部变量或常数的范围就是该块本身。

在块中，在表达式上下文中使用的名称的意义必须始终相同 (§ 7.5.2)。

块按下述规则执行：

- 如果块是空的，控制转移到块的结束点。
- 如果块不是空的，控制转移到语句列表。当（如果）控制到达语句列表的结束点时，控制转移到块的结束点。

如果块本身是可到达的，则块的语句列表是可到达的。

如果块是空的或者语句列表的结束点是可到达的，则块的结束点是可到达的。

语句列表由一个或多个按顺序编写的语句组成。语句列表出现在块 (§ 8.2) 和 `switch` 块 (§ 8.7.2) 中。

statement-list: (语句列表:)

statement (语句)

statement-list statement (语句列表 语句)

执行语句列表就是将控制转移到该列表中的第一个语句。当（如果）控制到达某条语句的结束点时，控制将转移到下一个语句。当（如果）控制到达最后一个语句的结束点时，控制将转移到语句列表的结束点。

如果下列条件中至少一个为真，则语句列表中的语句是可到达的：

- 该语句是第一个语句且语句列表本身是可到达的。
- 前一个语句的结束点是可到达的。

- 该语句本身是一个标记语句，并且该标签已被一个可到达的 `goto` 语句引用。如果列表中最后一个语句的结束点是可到达的，则语句列表的结束点是可到达的。

8.3 空语句

空语句什么都不做：

`empty-statement`: (空语句:)

;

当在要求有语句的上下文中不执行任何操作时，使用空语句。

执行空语句就是将控制转移到该语句的结束点。这样，如果空语句是可到达的，则空语句的结束点也是可到达的。

当编写一个语句体为空的 `while` 语句时，可以使用空语句：

```
bool ProcessMessage() {...}
void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

此外，空语句还可以用于在块的结束符 “}” 前声明标签：

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

8.4 标记语句

标记语句可以给语句加上一个标签作为前缀。标记语句可以出现在块中，但是不允许它们作为嵌入语句。

`labeled-statement`: (标记语句:)

`identifier` : `statement` (标识符 : 语句)

标记语句声明了一个标签，标签由一个标识符来命名。标签的范围为在其中声明了该标签的整个块，包括任何嵌套块。两个同名的标签若具有重叠的范围，则会产生一个编译时错误。

标签可以在该标签的范围内被 `goto` 语句 (§ 8.9.3) 引用。这意味着 `goto` 语句可以在它所在的块内转移控制，也可以将控制转移到该块外部，但是永远不能将控制转入该块所含的嵌套块的内部。

标签具有自己的声明空间，并不影响其他标识符。示例

```
int F(int x) {
    if (x >= 0) goto x;
```

```

        x = -x;
    x: return x;
}

```

是有效的，尽管它将 `x` 同时用做参数和标签的名称。

执行标记语句就是执行该标签后的那个语句。

除了由正常控制流程提供的可到达性外，如果一个标签由一个可到达的 `goto` 语句引用，则该标记语句是可到达的（例外的情形是：如果 `goto` 语句在一个包含了 `finally` 块的 `try` 中，标记语句本身在 `try` 之外，而且 `finally` 块的结束点不可到达，则从该 `goto` 语句不可到达上述的标记语句）。

8.5 声明语句

声明语句声明局部变量或常数。声明语句可以出现在块中，但不允许它们作为嵌入语句使用。

declaration-statement: (声明语句:)

local-variable-declaration ; (局部变量声明 ;)

local-constant-declaration ; (局部常数声明 ;)

8.5.1 局部变量声明

局部变量声明声明一个或多个局部变量。

local-variable-declaration: (局部变量声明:)

type **local-variable-declarators** (类型 局部变量声明符)

local-variable-declarators: (局部变量声明符:)

local-variable-declarator (局部变量声明符)

local-variable-declarators , **local-variable-declarator** (局部变量声明符 , 局部变量声明符)

local-variable-declarator: (局部变量声明符:)

identifier (标识符)

identifier = **local-variable-initializer** (标识符 = 局部变量初始值设定项)

local-variable-initializer: (局部变量初始值设定项:)

expression (表达式)

array-initializer (数组初始值设定项)

局部变量声明中的类型指定由该声明引入的变量类型。此类型后跟一个局部变量声明符列表，其中每个声明符引入一个新变量。局部变量声明符由命名变量的标识符组成，根据需要，此标识符后可跟一个“=”标记和一个赋予变量初始值的局部变量初始值设定项。

可以在表达式中通过简单名称 (§ 7.5.2) 来获取局部变量的值，还可以通过赋值 (§ 7.13) 来修改局部变量的值。在使用局部变量的每个地方必须先明确对其赋值 (§ 5.3)，然后才可以使用它的值。

在局部变量声明中声明的局部变量范围是该声明所在的块。在一个局部变量的局部变量声明符之前的文本位置中引用该局部变量是错误的。在一个局部变量的范围内声明其他具有相同名称的局部变量或常数将导致编译时错误。

声明了多个变量的局部变量声明等效于多个同一类型的单个变量的声明。另外，局部变量声明中的变量初始值设定项完全对应于紧跟该声明后插入的赋值语句。

示例

```
void F() {
    int x = 1, y, z = x * 2;
}
```

完全对应于

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

8.5.2 局部常数声明

局部常数声明用于声明一个或多个局部常数。

local-constant-declaration: (局部常数声明:)

const **type** **constant-declarators** (**const** 类型 常数声明符)

constant-declarators: (常数声明符:)

constant-declarator (常数声明符)

constant-declarators, constant-declarator (常数声明符, 常数声明符)

constant-declarator: (常数声明符:)

identifier = **constant-expression** (标识符 = 常数表达式)

局部常数声明的类型指定由该声明引入的常数的类型。此类型后跟一个常数声明符列表，其中每个常数声明符引入一个新常数。常数声明符包含一个命名常数的标识符，后跟一个“=”标记，然后是一个对该常数赋值的常数表达式 (§ 7.15)。

局部常数声明的类型和常数表达式必须遵循与常数成员声明 (§ 10.3) 一样的规则。

可以通过在表达式中使用一个简单名称 (§ 7.5.2) 来获取局部常数的值。

局部常数的范围是在其中声明了该常数的块。在某常数声明符之前的文本位置中引用该局部常数是错误的。在局部常数的范围内声明其他具有相同名称的局部变量或常数将导致编译时错误。

声明多个常数的局部常数声明等效于声明多个同一类型的单个常数。

8.6 表达式语句

表达式语句用于计算所给定的表达式。由此表达式计算出来的值（如果有的话）将被丢弃。

expression-statement: (表达式语句:)

statement-expression ; (语句表达式 ;)

statement-expression: (语句表达式:)

invocation-expression (调用表达式)

object-creation-expression (对象创建表达式)

assignment (赋值)

post-increment-expression (后递增表达式)

post-decrement-expression (后递减表达式)

pre-increment-expression (前递增表达式)

pre-decrement-expression (前递减表达式)

不是所有的表达式都可以作为语句使用。具体说来,不允许像 $x + y$ 和 $x == 1$ 这样只计算一个值(此值将被放弃)的表达式作为语句使用。

执行表达式语句就是对它所包含的表达式进行计算,然后将控制转移到该表达式语句的结束点。如果一个表达式语句是可到达的,那么其结束点也是可到达的。

8.7 选择语句

选择语句会根据表达式的值从若干个给定的语句中选择一个来执行。

selection-statement: (选择语句:)

if-statement (if 语句)

switch-statement (switch 语句)

8.7.1 if 语句

if 语句根据布尔表达式的值选择要执行的语句。

if-statement: (if 语句:)

if (**boolean-expression**) **embedded-statement** (if (布尔表达式) 嵌入语句)

if (**boolean-expression**) **embedded-statement** **else** **embedded-statement**
(if (布尔表达式) 嵌入语句 **else** 嵌入语句)

boolean-expression: (布尔表达式:)

expression (表达式)

else 部分与语法允许的、词法上最相近的上一个 if 语句相关联。因而,下列形式的 if 语句

```
if (x) if (y) F(); else G();
```

等效于

```
if (x) {  
    if (y) {
```



```

        F();
    }
    else {
        G();
    }
}

```

if 语句按如下规则执行：

- 计算布尔表达式 (§ 7.16)。
- 如果布尔表达式产生 true，则控制转移到第一个嵌入语句。当（如果）控制到达那条语句的结束点时，控制将转移到 if 语句的结束点。
- 如果布尔表达式产生 false 且如果存在 else 部分，则控制转移到第二个嵌入语句。当（如果）控制到达那条语句的结束点时，控制将转移到 if 语句的结束点。
- 如果布尔表达式产生 false 且如果不存在 else 部分，则控制转移到 if 语句的结束点。

如果 if 语句是可到达的且布尔表达式不具有常数值 false，则 if 语句的第一个嵌入语句是可到达的。

如果 if 语句是可到达的且布尔表达式不具有常数值 true，则 if 语句的第二个嵌入语句（如果存在的话）是可到达的。

如果 if 语句的至少一个嵌入语句的结束点是可到达的，则 if 语句的结束点是可到达的。此外，对于不具有 else 部分的 if 语句，如果 if 语句是可到达的且布尔表达式不具有常数值 true，则该 if 语句的结束点是可到达的。

8.7.2 switch 语句

switch 语句选择一个要执行的语句列表，此列表具有一个相关联的 switch 标签，它对应于 switch 表达式的值。

switch-statement: (switch 语句:)

switch (expression) switch-block (switch (表达式)
switch 块)

switch-block: (switch 块:)

{ switch-sections_{opt} } ({ switch 节_{可选} })

switch-sections: (switch 节:)

switch-section (switch 节)

switch-sections switch-section (switch 节 switch 节)

switch-section: (switch 节:)

switch-labels statement-list (switch 标签 语句列表)

switch-labels: (switch 标签:)

switch-label (switch 标签)

switch-labels switch-label (switch 标签 switch 标签)

switch-label: (switch 标签:)

```

    case    constant-expression    : (case    常数表达式    :)
    default : (default    :)

```

switch 语句包含关键字 switch，后跟带括号的表达式（称为 switch 表达式），然后是一个 switch 块。switch 块包含零个或多个括在大括号内的 switch 节。每个 switch 节包含一个或多个 switch 标签，后跟一个语句列表（§ 8.2）。

switch 语句的**主导类型（governing type）**由 switch 表达式建立。如果 switch 表达式的类型为 sbyte、byte、short、ushort、int、uint、long、ulong、char、string 或枚举类型，那么这就是 switch 语句的主导类型。否则，必须有且只有一个用户定义的从 switch 表达式的类型到下列某个可能的主导类型的隐式转换（§ 6.4）：sbyte, byte, short, ushort, int, uint, long, ulong, char, string。如果不存在这样的隐式转换，或者存在多于一个这样的隐式转换，则发生编译时错误。

每个 case 标签的常数表达式的值必须属于这种类型：它可以隐式转换（§ 6.1）为 switch 语句的主导类型。如果同一 switch 语句中的两个或更多个 case 标签指定同一个常数值，则导致编译时错误。

一个 switch 语句中最多只能有一个 default 标签。

switch 语句按如下规则执行：

- 计算 switch 表达式并将其转换为主导类型。
- 如果在该 switch 语句的 case 标签中，有一个指定的常数恰好等于 switch 表达式的值，控制将转移到匹配的 case 标签后的语句列表。
- 如果在该 switch 语句的 case 标签中，指定的常数都不等于 switch 表达式的值，并且如果存在一个 default 标签，则控制将转移到 default 标签后的语句列表。
- 如果在该 switch 语句的 case 标签中，指定的常数都不等于 switch 表达式的值，并且如果不存在 default 标签，则控制将转移到 switch 语句的结束点。

如果 switch 节的语句列表的结束点是可达的，将发生编译时错误。这称为“非贯穿”规则。示例

```

switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
default:
    CaseOthers();
    break;
}

```

是有效的，这是因为没有一个 switch 节的结束点是可达的。与 C 和 C++ 不同，执行一个 switch 节的过程不能“贯穿”到下一个 switch 节，示例

```

switch (i) {
case 0:
    CaseZero();
case 1:

```

```

    CaseZeroOrOne();
default:
    CaseAny();
}

```

将导致编译时错误。如果要在执行一个 `switch` 节后继续执行另一个 `switch` 节，则必须使用显式的 `goto case` 语句 或 `goto default` 语句。

```

switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}

```

在一个 `switch` 节中允许有多个标签。示例

```

switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
    break;
}

```

是有效的。此示例不违反“非贯穿”规则，这是因为标签 `case 2:` 和 `default:` 属于同一个 `switch` 节。

“非贯穿”规则防止了在 C 和 C++ 中由于不经意地漏掉 `break` 语句而引起的一类常见错误。另外，由于这个规则，`switch` 语句的各个 `switch` 节可以任意重新排列而不会影响语句的行为。例如，上面 `switch` 语句中的各节的顺序可以在不影响语句行为的情况下反转排列：

```

switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}

```

`switch` 节的语句列表通常以 `break` 语句，`goto case` 语句 或 `goto default` 语句结束，但是也可以使用任何其他结构，只要能保证对应的语句列表的结束点是不可到达的。例如，

由布尔表达式 `true` 控制的 `while` 语句是永远无法到达其结束点的。同样，`throw` 语句 或 `return` 语句始终将控制转移到其他地方而从不到达它的结束点。因此，下面的示例是有效的：

```
switch (i) {
    case 0:
        while (true) F();
    case 1:
        throw new ArgumentException();
    case 2:
        return;
}
```

`switch` 语句的主导类型可以是 `string` 类型。例如：

```
void DoCommand(string command) {
    switch (command.ToLower()) {
        case "run":
            DoRun();
            break;
        case "save":
            DoSave();
            break;
        case "quit":
            DoQuit();
            break;
        default:
            InvalidCommand(command);
            break;
    }
}
```

与字符串相等运算符 (§ 7.9.7) 一样，`switch` 语句区分大小写，因而只有在 `switch` 表达式字符串与 `case` 标签常数完全匹配时才会执行对应的 `switch` 节。

当 `switch` 语句的主导类型为 `string` 时，允许值 `null` 作为 `case` 标签常数。

`switch` 块的语句列表可以包含声明语句 (§ 8.5)。在 `switch` 块中声明的局部变量或常数的范围是该 `switch` 块。

在 `switch` 块内，表达式上下文中使用的名称的意义必须始终是相同的 (§ 7.5.2.1)。

如果 `switch` 语句是可到达的且下列条件至少有一个为真，则给定的 `switch` 节的语句列表是可到达的：

- `switch` 表达式是非常数值。
- `switch` 表达式是与该 `switch` 节中的某个 `case` 标签匹配的常数值。
- `switch` 表达式是不与任何 `case` 标签匹配的常数值，且该 `switch` 节包含 `default` 标签。
- 该 `switch` 节的某个 `switch` 标签由一个可到达的 `goto case` 语句或 `goto default` 语句引用。

如果下列条件中至少有一个为真，则 `switch` 语句的结束点是可达的：

- `switch` 语句包含一个退出 `switch` 语句的可到达的 `break` 语句。
- `switch` 语句是可到达的，`switch` 表达式是非常数值，并且不存在 `default` 标签。
- `switch` 语句是可到达的，`switch` 表达式是不与任何 `case` 标签匹配的常数值，并

且不存在任何 default 标签。

8.8 迭代语句

迭代语句重复执行嵌入语句。

iteration-statement: (迭代语句:)

while-statement (while 语句)

do-statement (do 语句)

for-statement (for 语句)

foreach-statement (foreach 语句)

8.8.1 while 语句

while 语句按不同条件执行一个嵌入语句零次或多次。

while-statement: (while 语句:)

while (boolean-expression) embedded-statement (while (布尔表达式) 嵌入语句)

while 语句按如下规则执行:

- 计算布尔表达式 (§ 7.16)。
- 如果布尔表达式产生 true, 控制将转移到嵌入语句。当 (如果) 控制到达嵌入语句的结束点 (可能是通过执行 continue 语句) 时, 控制将转移到 while 语句的开头。
- 如果布尔表达式产生 false, 控制将转移到 while 语句的结束点。

在 while 语句的嵌入语句内, break 语句 (§ 8.9.1) 可用于将控制转移到 while 语句的结束点 (从而结束嵌入语句的迭代), 而 continue 语句 (§ 8.9.2) 可用于将控制转移到嵌入语句的结束点。

如果 while 语句是可到达的且布尔表达式不具有常数值 false, 则该 while 语句的嵌入语句是可到达的。

如果下列条件中至少有一个为真, 则 while 语句的结束点是可到达的:

- while 语句包含一个可到达的 break 语句 (它用于退出 while 语句)。
- while 语句是可到达的且布尔表达式不具有常数值 true。

8.8.2 do 语句

do 语句按不同条件执行一个嵌入语句一次或多次。

do-statement: (do 语句:)

do embedded-statement while (boolean-expression); (do 嵌入语句 while (布尔表达式);)

do 语句按如下规则执行：

- 控制转移到嵌入语句。
- 当（如果）控制到达嵌入语句的结束点（可能是通过执行一个 continue 语句）时，计算布尔表达式（§ 7.16）。如果布尔表达式产生 true，控制将转移到 do 语句的开头。否则，控制转移到 do 语句的结束点。

在 do 语句的嵌入语句内，break 语句（§ 8.9.1）可用于将控制转移到 do 语句的结束点（从而结束嵌入语句的迭代），而 continue 语句（§ 8.9.2）可用于将控制转移到嵌入语句的结束点（从而执行 do 语句的另一次迭代）。

如果 do 语句是可到达的，则 do 语句的嵌入语句是可到达的。

如果下列条件中至少有一个为真，则 do 语句的结束点是可达的：

- do 语句包含一个可达的 break 语句（它用于退出 do 语句）。
- 嵌入语句的结束点是可达的且布尔表达式不具有常数值 true。

8.8.3 for 语句

for 语句计算一个初始化表达式序列，然后，当某个条件为真时，重复执行相关的嵌套语句并计算一个迭代表达式序列。

for-statement: (for 语句:)

for (for-initializer_{opt} ; for-condition_{opt} ; for-iterator_{opt}) embedded-statement (for (for 初始值设定项_{可选} ; for 条件_{可选} ; for 迭代程序_{可选}) 嵌入语句)

for-initializer: (for 初始值设定项:)

local-variable-declaration (局部变量声明)

statement-expression-list (语句表达式列表)

for-condition: (for 条件:)

boolean-expression (布尔表达式)

for-iterator: (for 迭代程序:)

statement-expression-list (语句表达式列表)

statement-expression-list: (语句表达式列表:)

statement-expression (语句表达式)

statement-expression-list , statement-expression (语句表达式列表 , 语句表达式)

for 初始值设定项（如果存在的话）或者由一个局部变量声明（§ 8.5.1），或者由一个用逗号分隔的语句表达式（§ 8.6）列表组成。用 for 初始值设置项声明的局部变量的范围从变量的局部变量声明符开始，一直延伸到嵌入语句的结尾。该范围包括 for 条件和 for 迭代程序。

for 条件（如果存在的话）必须是布尔表达式（§ 7.16）。

for 迭代程序（如果存在的话）包含一个用逗号分隔的语句表达式（§ 8.6）列表。

for 语句按如下规则执行：

- 如果存在 for 初始值设定项，则按变量初始值设定项或语句表达式的编写顺序执

行它们。此步骤只执行一次。

- 如果存在 for 条件, 则计算它。
- 如果不存在 for 条件或计算产生 true, 控制将转移到嵌入语句。当(如果)控制到达嵌入语句的结束点(可能是通过执行一个 continue 语句)时, 按顺序计算 for 迭代程序的表达式(如果有的话), 然后从上述步骤中的计算 for 条件开始, 执行另一次迭代。
- 如果存在 for 条件, 且计算产生 false, 控制将转移到 for 语句的结束点。

在 for 语句的嵌入语句内, break 语句 (§ 8.9.1) 可用于将控制转移到 for 语句的结束点, 从而结束嵌入语句的迭代; 而 continue 语句 (§ 8.9.2) 可用于将控制转移到嵌入语句的结束点, 从而执行 for 迭代程序并从 for 条件开始执行 for 语句的另一次迭代。

如果下列条件之一为真, 则 for 语句的嵌入语句是可到达的:

- for 语句是可到达的且不存在 for 条件。
- for 语句是可到达的, 并且存在一个 for 条件, 它不具有常数值 false。

如果下列条件中至少有一个为真, 则 for 语句的结束点是可达的:

- for 语句包含一个可达的 break 语句(它用于退出 for 语句)。
- for 语句是可到达的, 并且存在一个 for 条件, 它不具有常数值 true。

8.8.4 foreach 语句

foreach 语句用于枚举一个集合的元素, 并对该集合中的每个元素执行一次相关的嵌入语句。

foreach-statement: (foreach 语句:)

foreach (type identifier in expression) embedded-statement (foreach (类型标识符 in 表达式) 嵌入语句)

foreach 语句的类型和标识符声明该语句的**迭代变量 (iteration variable)**。迭代变量相当于一个其范围覆盖整个嵌入语句的只读局部变量。在 foreach 语句执行期间, 迭代变量表示当前正在为其执行迭代的集合元素。如果嵌入语句试图修改迭代变量(通过赋值或 ++ 和 -- 运算符)或将迭代变量作为 ref 或 out 参数传递, 则将发生编译时错误。

foreach 语句的表达式必须是集合类型(下面有它的定义), 且必须有一个从该集合的元素类型到迭代变量的类型的显式转换 (§ 6.2)。如果该表达式具有 null 值, 则将引发 System.NullReferenceException。

如果类型 C 实现了 System.Collections.IEnumerable 接口, 或者能够满足下列(有关实现集合模式的)条件, 就称它是**集合类型 (collection type)**。

- C 包含一个 public 实例方法, 它带有签名 GetEnumerator(), 且返回值属于结构类型、类类型或接口类型(后文中用 E 表示该返回值的类型)。
- E 包含一个 public 实例方法, 此方法具有签名 MoveNext() 和返回类型 bool。
- E 包含一个名为 Current 的 public 实例属性, 此属性允许读取当前值。此属性的类型称为该集合类型的元素类型。

一个实现 IEnumerable 的类型也是集合类型, 即使它不满足上述条件(这是可能的,

如果该类型通过“显式接口成员实现”的方式实现某些 `IEnumerable` 成员，关于该方式详见第 13.4.1 节中的说明)。

`System.Array` 类型 (§ 12.1) 是集合类型，而由于所有的数组类型都从 `System.Array` 派生，因此 `foreach` 语句中允许使用任何数组类型表达式。`foreach` 按如下顺序遍历数组的元素：对于一维数组，按递增的索引顺序遍历元素，从索引 0 开始，到索引 `Length - 1` 结束。对于多维数组，按这样的方式遍历元素：首先增加最右边维度的索引，然后是它的左边紧邻的维度，依此类推直到最左边的那个维度。

下列形式的 `foreach` 语句：

```
foreach (ElementType element in collection) statement
```

对应于两个可能的扩展中的一个。

- 如果 `collection` 表达式的类型实现了集合模式（如上面定义的那样），则 `foreach` 语句的扩展是：

```
E enumerator = (collection).GetEnumerator();
try {
    while (enumerator.MoveNext()) {
        ElementType element = (ElementType)enumerator.Current;
        statement;
    }
}
finally {
    IDisposable disposable = enumerator as System.IDisposable;
    if (disposable != null) disposable.Dispose();
}
```

通常可以很容易地对上述代码进行有效的优化。如果类型 `E` 实现了 `System.IDisposable`，则表达式 `(enumerator as System.IDisposable)` 将始终为非空，因而，该实现可以安全地用一个简单转换替代可能更昂贵的类型测试。相反，如果类型 `E` 是封闭的而且没有实现 `System.IDisposable`，则表达式 `(enumerator as System.IDisposable)` 的计算结果始终为 `null`。这种情况下，在上述代码中可以安全地删除整个 `finally` 子句。

- 否则，`collection` 表达式的类型就实现了 `System.IEnumerable`，这样，`foreach` 语句的扩展就成为：

```
IEnumerator enumerator =
    ((System.Collections.IEnumerable)(collection)).GetEnumerator();
try {
    while (enumerator.MoveNext()) {
        ElementType element = (ElementType)enumerator.Current;
        statement;
    }
}
finally {
    IDisposable disposable = enumerator as System.IDisposable;
    if (disposable != null) disposable.Dispose();
}
```

在上面两个扩展的任意一个中，`enumerator` 变量是一个临时变量，它在嵌入式语句中既是不可访问的，也是不可见的，元素变量在嵌入式语句中是只读的。

下列示例按照元素的顺序打印出一个二维数组中的各个元素的值：

```
using System;
class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };
        foreach (double elementValue in values)
            Console.WriteLine(elementValue);
    }
}
```

所生成的输出如下：

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

8.9 跳转语句

跳转语句用于无条件地转移控制。

jump-statement: (跳转语句:)

break-statement (break 语句)

continue-statement (continue 语句)

goto-statement (goto 语句)

return-statement (return 语句)

throw-statement (throw 语句)

跳转语句会将控制转移到某个位置，这个位置称为跳转语句的目标。

当一个跳转语句出现在某个块内，而该跳转语句的目标在该块之外时，就称该跳转语句退出该块。虽然跳转语句可以将控制转移到一个块外，但它永远不能将控制转移到一个块的内部。

由于存在 `try` 语句的干扰，跳转语句的执行有时会变得复杂起来。如果没有这样的 `try` 语句，则跳转语句无条件地将控制从跳转语句转移到它的目标。当跳转涉及 `try` 语句时，执行就变得复杂一些了。如果跳转语句欲退出的是一个或多个具有相关联的 `finally` 块的 `try` 块，则控制最初转移到最里层的 `try` 语句的 `finally` 块。当（如果）控制到达该 `finally` 块的结束点时，控制就转移到下一个封闭的 `try` 语句中的 `finally` 块。此过程不断重复，直到执行完所有涉及 `try` 语句的 `finally` 块。

在下面的示例中，在将控制转移到跳转语句的目标之前，要先执行与两个 `try` 语句关联的 `finally` 块。

```
using System;
class Test
{
    static void Main() {
        while (true) {
            try {
```

```
        try {
            Console.WriteLine("Before break");
            break;
        }
        finally {
            Console.WriteLine("Innermost finally block");
        }
    }
    finally {
        Console.WriteLine("Outermost finally block");
    }
}
Console.WriteLine("After break");
}
```

所生成的输出如下:

```
Before break
Innermost finally block
Outermost finally block
After break
```

8.9.1 break 语句

break 语句退出直接封闭着它的 **switch**, **while**, **do**, **for** 或 **foreach** 语句。

break-statement: (**break** 语句:)

```
break ;
```

break 语句的目标是直接封闭着它的 **switch**, **while**, **do**, **for** 或 **foreach** 语句的结束点。如果 **break** 语句不是由 **switch**、**while**、**do**、**for** 或 **foreach** 语句所封闭, 则发生编译时错误。

当多个 **switch**, **while**, **do**, **for** 或 **foreach** 语句彼此嵌套时, **break** 语句只应用于最里层的语句。若要穿越多个嵌套层直接转移控制, 必须使用 **goto** 语句 (§ 8.9.3)。

break 语句不能退出 **finally** 块 (§ 8.10)。当 **break** 语句出现在 **finally** 块中时, 该 **break** 语句的目标必须位于同一个 **finally** 块中, 否则将发生编译时错误。

break 语句按如下规则执行:

- 如果 **break** 语句退出一个或多个具有关联 **finally** 块的 **try** 块, 则控制最初会被转移到最里层的 **try** 语句的 **finally** 块。当(如果)控制到达该 **finally** 块的结束点时, 控制就转移到下一个封闭 **try** 语句的 **finally** 块。此过程不断重复, 直到执行完所有涉及 **try** 语句的 **finally** 块。
- 控制转移到 **break** 语句的目标。

由于 **break** 语句无条件地将控制转移到别处, 因此永远无法到达 **break** 语句的结束点。

8.9.2 continue 语句

continue 语句开始直接封闭着它的 while, do, for 或 foreach 语句的一次新迭代。

continue-statement: (continue 语句:)

```
continue ;
```

continue 语句的目标是直接封闭着它的 while, do, for 或 foreach 语句的嵌套语句的结束点。如果 continue 语句不是由 while, do, for 或 foreach 语句所封闭的, 则发生编译时错误。

当多个 while, do, for 或 foreach 语句互相嵌套时, continue 语句只应用于最里层的那个语句。若要穿越多个嵌套层直接转移控制, 必须使用 goto 语句 (§ 8.9.3)。

continue 语句不能退出 finally 块 (§ 8.10)。当 continue 语句出现在 finally 块中时, 该 continue 语句的目标必须位于同一个 finally 块中, 否则将发生编译时错误。

continue 语句按如下规则执行:

- 如果 continue 语句退出一个或多个具有关联 finally 块的 try 块, 则控制最初转移到最里层的 try 语句的 finally 块。当(如果)控制到达该 finally 块的结束点时, 控制就转移到下一个封闭 try 语句的 finally 块。此过程不断重复, 直到执行完所有涉及 try 语句的 finally 块。
- 控制转移到 continue 语句的目标。

由于 continue 语句无条件地将控制转移到别处, 因此永远无法到达 continue 语句的结束点。

8.9.3 goto 语句

goto 语句将控制转移到由标签标记的语句。

goto-statement: (goto 语句:)

```
goto identifier ; (goto 标识符 ;)
goto case constant-expression ; (goto case 常数表达式 ;)
goto default ; (goto default ;)
```

goto 标识符语句的目标是具有给定标签的标记语句。如果当前函数成员中不存在具有给定名称的标签, 或者 goto 语句不在该标签的范围内, 则发生编译时错误。此规则允许使用 goto 语句将控制转移“出”嵌套范围, 但是不允许将控制转移“进”嵌套范围。在下面的示例中, goto 语句用于将控制转移出嵌套范围。

```
using System;
class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };
    }
}
```

```

foreach (string str in args) {
    int row, colm;
    for (row = 0; row <= 1; ++row)
        for (colm = 0; colm <= 2; ++colm)
            if (str == table[row,colm])
                goto done;
    Console.WriteLine("{0} not found", str);
    continue;
done:
    Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
}
}

```

goto case 语句的目标是直接封闭着它的 **switch** 语句 (§ 8.7.2) 中的某个语句列表，此列表包含一个具有给定常数值 **case** 标签。如果 **goto case** 语句不是由 **switch** 语句封闭的，或者常数表达式不能隐式转换 (§ 6.1) 为直接封闭着它的 **switch** 语句的主导类型，或者直接封闭着它的 **switch** 语句不包含具有给定常数值 **case** 标签，则发生编译时错误。

goto default 语句的目标是直接封闭着它的那个 **switch** 语句 (§ 8.7.2) 中对应于 **default** 标签的语句列表。如果 **goto default** 语句不是由 **switch** 语句封闭的，或者直接封闭着它的 **switch** 语句不包含 **default** 标签，则发生编译时错误。

goto 语句不能退出 **finally** 块 (§ 8.10)。当 **goto** 语句出现在 **finally** 块中时，该 **goto** 语句的目标必须位于同一个 **finally** 块中，否则将发生编译时错误。

goto 语句按如下方式执行：

- 如果 **goto** 语句退出一个或多个具有关联的 **finally** 块的 **try** 块，则控制最初转移到最里层的 **try** 语句的 **finally** 块。当(如果)控制到达 **finally** 块的结束点时，控制就转移到下一个封闭 **try** 语句的 **finally** 块。此过程不断重复，直到执行完所有涉及 **try** 语句的 **finally** 块。
- 控制转移到 **goto** 语句的目标。

由于 **goto** 语句无条件地将控制转移到别处，因此永远无法到达 **goto** 语句的结束点。

8.9.4 return 语句

return 语句将控制返回到出现 **return** 语句的函数成员的调用方。

return-statement: (**return** 语句:)

return **expression**_{opt} ; (**return** 表达式_{可选} ;)

不带表达式的 **return** 语句只能用在不计算值的函数成员中，即只能用在返回类型为 **void** 的方法、属性或索引器的 **set** 访问器、事件的 **add** 和 **remove** 访问器、实例构造函数、静态构造函数或析构函数中。

带表达式的 **return** 语句只能用在计算值的函数成员中，即返回类型为非 **void** 的方法、属性或索引器的 **get** 访问器或用户定义的运算符。必须存在一个隐式转换 (§ 6.1)，它能将该表达式的类型转换到包含它的函数成员的返回类型。

return 语句出现在 **finally** 块 (§ 8.10) 中会出现编译时错误。

return 语句按下列规则执行:

- 如果 return 语句指定一个表达式, 则计算该表达式, 并将结果隐式转换为包含它的函数成员的返回类型。转换的结果成为返回到调用方的值。
- 如果 return 语句由一个或多个具有关联 finally 块的 try 块封闭, 则控制最初转移到最里层的 try 语句的 finally 块。当(如果)控制到达 finally 块的结束点时, 控制转移到下一个封闭 try 语句的 finally 块。此过程不断重复, 直到执行完所有封闭 try 语句的 finally 块。
- 控制返回到包含它们的函数成员的调用方。

由于 return 语句无条件地将控制转移到别处, 因此永远无法到达 return 语句的结束点。

8.9.5 throw 语句

throw 语句抛出一个异常。

throw-statement: (throw 语句:)

throw expression_{opt} ; (throw 表达式_{可选} ;)

带表达式的 throw 语句抛出通过计算该表达式而产生的值。计算该表达式所得的值必须属于类类型 System.Exception 或从 System.Exception 派生的类类型。如果表达式的计算产生 null, 则抛出 System.NullReferenceException。

不带表达式的 throw 语句只能用在 catch 块中, 在这种情况下, 该语句重新抛出当前正由该 catch 块处理的那个异常。

由于 throw 语句无条件地将控制转移到别处, 因此永远无法到达 throw 语句的结束点。

抛出一个异常时, 控制转移到封闭着它的 try 语句中能够处理该异常的的第一个 catch 子句。从抛出一个异常开始直至将控制转移到关于该异常的一个合适的异常处理程序为止, 这个过程称为异常传播。“传播一个异常”由重复地执行下列各步骤组成, 直至找到一个与该异常匹配的 catch 子句。在此描述中, 抛出点最初是指抛出该异常的位置。

- 在当前函数成员中, 检查每个封闭着抛出点的 try 语句。对于每个语句 S (顺序是从最里层的 try 语句开始, 逐次向外, 直到最外层的 try 语句结束), 计算下列步骤:
 - 如果 S 的 try 块封闭着抛出点, 并且 S 具有一个或多个 catch 子句, 则按其出现的顺序检查这些 catch 子句以找到合适的异常处理程序。检查过程中, 第一个符合要求的 catch 子句 (它所指定的异常类型与该异常的类型相同或是它的基类型) 被认为是一个匹配项。常规 catch 子句 (§ 8.10) 被认为是所有异常类型的匹配项。如果找到匹配的 catch 子句, 则通过将控制转移到该 catch 子句的块来完成异常传播。
 - 否则 (如果找不到匹配的 catch 子句), 如果 S 的 try 块或 catch 块封闭着抛出点并且如果 S 具有 finally 块, 控制将转移到 finally 块。如果在该 finally 块内抛出另一个异常, 则终止对当前异常的处理。否则, 当控制到达 finally 块的

结束点时，将继续对当前异常的处理。

- 如果在当前函数成员调用中没有找到异常处理程序，则终止对该函数成员的调用。然后，为该函数成员的调用方重复执行上面的步骤，并使用对应于该调用函数成员的语句的抛出点。
- 如果上述异常处理终止了当前线程中的所有函数成员调用，这表明此线程没有该异常的处理程序，那么线程本身将终止。此类终止会产生什么影响，应由实现来定义。

8.10 try 语句

try 语句提供一种机制，用于捕获在块的执行期间发生的各种异常。此外，try 语句还能让你指定一个代码块，并保证当控制离开 try 语句时，先执行该代码。

try-statement: (try 语句:)

try block catch-clauses (try 块 catch 子句)

try block finally-clause (try 块 finally 子句)

try block catch-clauses finally-clause (try 块 catch 子句 finally 子句)

catch-clauses: (catch 子句:)

specific-catch-clauses general-catch-clauseopt (特定 catch 子句 常规 catch 子句可选)

specific-catch-clausesopt general-catch-clause (特定 catch 子句可选 常规 catch 子句)

specific-catch-clauses: (特定 catch 子句:)

specific-catch-clause (特定 catch 子句)

specific-catch-clauses specific-catch-clause (特定 catch 子句 特定 catch 子句)

specific-catch-clause: (特定 catch 子句:)

catch (class-type identifier_{opt}) block (catch (类类型 标识符_{可选}) 块)

general-catch-clause: (常规 catch 子句:)

catch block (catch 块)

finally-clause: (finally 子句:)

finally block (finally 块)

有三种可能的 try 语句形式:

- 一个 try 块后跟一个或多个 catch 块。
- 一个 try 块后跟一个 finally 块。
- 一个 try 块后跟一个或多个 catch 块，后面再跟一个 finally 块。

当 catch 子句指定类类型时，此类型必须为 System.Exception 或从 System.Exception

派生的类型。

当 `catch` 子句同时指定类类型和标识符时，相当于声明了一个具有给定名称和类型的异常变量。此异常变量相当于一个范围覆盖整个 `catch` 块的局部变量。在 `catch` 块的执行期间，此异常变量表示当前正在处理的异常。出于明确赋值检查的目的，此异常变量被认为在它的整个范围内是明确赋值的。

除非 `catch` 子句包含一个异常变量名，否则在该 `catch` 块中就不可能访问当前发生的异常对象。

既不指定异常类型，也不指定异常变量名的 `catch` 子句称为常规 `catch` 子句。一个 `try` 语句只能有一个常规 `catch` 子句，而且它必须是最后一个 `catch` 子句（因此如果存在的话）。

有些编程语言可能支持一些异常，它们不能表示为从 `System.Exception` 派生的对象，尽管 C# 代码可能永远不会产生这类异常。可以使用常规 `catch` 子句来捕获这类异常。因此，常规的 `catch` 子句在语义上不同于指定了 `System.Exception` 类型的那些子句，因为前者还可以捕获来自其他语言的异常。

为了找到当前发生了的异常的处理程序，`catch` 子句是按其词法顺序进行检查的。如果 `catch` 子句指定的类型与同一 `try` 块的某个较早的 `catch` 子句中所指定的类型相同，或者是从该类型派生的类型，则发生编译时错误。如果没有这个限制，就可能写出不可到达的 `catch` 子句。

在 `catch` 块内，不具有表达式的 `throw` 语句（§ 8.9.5）可用于重新抛出由该 `catch` 块捕获的异常。对异常变量的赋值不会改变上述被重新抛出的异常。

下面是一个示例：

```
using System;
class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw;          // 重新抛出
        }
    }
    static void G() {
        throw new Exception("G");
    }
    static void Main() {
        try {
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

示例中，方法 `F` 捕获一个异常，向控制台写入一些诊断信息，更改异常变量，然后

重新抛出该异常。重新抛出的异常是原来那个被捕获的异常，因此产生的输出为：

```
Exception in F: G  
Exception in Main: G
```

如果修改第一个 `catch` 块中的代码，用抛出异常 `e` 来代替重新抛出当前的异常，产生的输出就会如下所示：

```
Exception in F: G  
Exception in Main: F
```

`break`、`continue` 或 `goto` 语句将控制转移到 `finally` 块外部会发生编译时错误。当 `break`、`continue` 或 `goto` 语句出现在 `finally` 块中时，该语句的目标必须在同一个 `finally` 块内，否则会发生编译时错误。

`return` 语句出现在 `finally` 块中会发生编译时错误。

`try` 语句按下列规则执行。

- 控制转移到 `try` 块。
- 当（如果）控制到达 `try` 块的结束点时：
 - ◆ 如果 `try` 语句具有 `finally` 块，则执行 `finally` 块。
 - ◆ 控制转移到 `try` 语句的结束点。
- 如果在 `try` 块执行期间，有一个异常传播到该 `try` 语句：
 - ◆ 按 `catch` 子句出现的顺序（如果有的话）逐个对其进行检查，以找到一个合适的异常处理程序。检查过程中，第一个符合要求的 `catch` 子句（它所指定的异常类型与该异常的类型相同或是它的基类型）被认为是一个匹配项。常规 `catch` 子句被认为是任何异常类型的匹配项。如果找到匹配的 `catch` 子句，则按下面这样处理。
 - 如果匹配的 `catch` 子句声明一个异常变量，则当前的异常对象被赋给该异常变量。
 - 控制转移到匹配的 `catch` 块。
 - 当（如果）控制到达 `catch` 块的结束点时，按以下规则处理。
 - 如果 `try` 语句具有 `finally` 块，则执行 `finally` 块。
 - 控制转移到 `try` 语句的结束点。
 - 如果在 `catch` 块执行期间有一个异常传播到 `try` 语句，则按以下规则处理。
 - 如果该 `try` 语句具有 `finally` 块，则执行 `finally` 块。
 - 该异常传播到下一个封闭 `try` 语句。
 - ◆ 如果该 `try` 语句没有 `catch` 子句或如果没有与异常匹配的 `catch` 子句，则按以下规则处理。
 - 如果该 `try` 语句具有 `finally` 块，则执行 `finally` 块。
 - 该异常就传播到更外面一层（封闭）的 `try` 语句。

`finally` 块中的语句总是在控制离开 `try` 语句时先被执行。无论是什么原因引起控制转移（正常执行到达结束点，执行了 `break`、`continue`、`goto` 或 `return` 语句，或是执行一个

导致离开 `try` 语句的异常传播), 情况都是如此。

如果在执行 `finally` 块期间抛出了一个异常, 则此异常将被传播到下一个封闭的 `try` 语句。与此同时, 原先那个正在传播过程中的异常(如果存在的话)就会被丢弃。关于传播异常的过程, 在 `throw` 语句 (§ 8.9.5) 的说明中有进一步讨论。

如果 `try` 语句是可到达的, 则 `try` 语句的 `try` 块是可到达的。

如果 `try` 语句是可到达的, 则 `try` 语句的 `catch` 块是可到达的。

如果 `try` 语句是可到达的, 则 `try` 语句的 `finally` 块是可到达的。

如果下列两个条件都为真, 则 `try` 语句的结束点是可到达的:

- `try` 块的结束点是可到达的或者至少一个 `catch` 块的结束点是可到达的。
- 如果存在一个 `finally` 块, 此 `finally` 块的结束点是可到达的。

8.11 checked 语句和 unchecked 语句

`checked` 语句和 `unchecked` 语句用于控制整型算术运算和转换的溢出检查上下文。

`checked-statement`: (`checked` 语句:)

`checked` `block` (`checked` 块)

`unchecked-statement`: (`unchecked` 语句:)

`unchecked` `block` (`unchecked` 块)

`checked` 语句导致它指定的块中的所有表达式都在一个选中的上下文中进行计算, 而 `unchecked` 语句导致它们在一个未选中的上下文中进行计算。

`checked` 语句和 `unchecked` 语句完全等效于 `checked` 运算符和 `unchecked` 运算符 (§ 7.5.12), 不同的只是它们作用于块, 而不是表达式。

8.12 lock 语句

`lock` 语句用于获取某个给定对象的互斥锁, 执行一个语句, 然后释放该锁。

`lock-statement`: (`lock` 语句:)

`lock` (`expression`) `embedded-statement` (`lock` (`表达式`)
嵌入语句)

`lock` 语句的表达式必须表示一个引用类型的值。永远不会为 `lock` 语句中的表达式执行隐式装箱转换 (§ 6.1.5), 因此, 如果该表达式表示的是一个值类型的值, 则会导致编译时错误。

下列形式的 `lock` 语句 (其中 `x` 是一个引用类型的表达式)

`lock (x) ...`

完全等效于

```
System.Threading.Monitor.Enter(x);
try {
    ...
}
```

```

    }
    finally {
        System.Threading.Monitor.Exit(x);
    }
}

```

不同的只是实际执行中 `x` 只计算一次。

当一个互斥锁已被占用时，在同一线程中执行的代码仍可以获取和释放该锁。但是，在其他线程中执行的代码在该锁被释放前是无法获取它的。

类的 `System.Type` 对象可以方便地用来当做关于该类的静态方法的互斥锁。例如：

```

class Cache
{
    public static void Add(object x) {
        lock (typeof(Cache)) {
            ...
        }
    }
    public static void Remove(object x) {
        lock (typeof(Cache)) {
            ...
        }
    }
}

```

8.13 using 语句

`using` 语句获取一个或多个资源，执行一个语句，然后处置该资源。

using-statement: (`using` 语句:)

```

using ( resource-acquisition ) embedded-statement ( using
( 资源获取 ) 嵌入语句)

```

resource-acquisition: (资源获取:)

`local-variable-declaration` (局部变量声明)

`expression` (表达式)

资源(resource)是实现了 `System.IDisposable` 的类或结构，它只包含一个名为 `Dispose` 的不带参数的方法。正在使用资源的代码可以调用 `Dispose` 以表明不再需要该资源。如果不调用 `Dispose`，则最终将由于垃圾回收而对该资源进行自动处置。

如果资源获取的形式是局部变量声明，那么此局部变量声明的类型必须为 `System.IDisposable` 或可以隐式转换为 `System.IDisposable` 的类型。如果资源获取的形式是表达式，那么此表达式必须是 `System.IDisposable` 类型或可以隐式转换为 `System.IDisposable` 的类型。

在资源获取中声明的局部变量是只读的，并且必须包含一个初始值设定项。如果嵌入语句试图修改这些局部变量(通过赋值或 `++` 和 `--` 运算符)或将它们作为 `ref` 或 `out` 参数传递，则将发生编译时错误。

`using` 语句被翻译成三个部分：获取、使用和处置。资源的使用部分被隐式封闭在一个含有 `finally` 子句的 `try` 语句中，此 `finally` 子句用于处置资源。如果所获取资源是 `null`，则不会对 `Dispose` 进行调用，也不会引发任何异常。

下列形式的 `using` 语句

```
using (ResourceType resource = expression) statement
```

对应于下列两个可能的扩展中的一个。当 `ResourceType` 是值类型时，扩展为

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        ((IDisposable)resource).Dispose();
    }
}
```

否则，当 `ResourceType` 是引用类型时，扩展为

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        if (resource != null) ((IDisposable)resource).Dispose();
    }
}
```

在上面任何一种扩展中，`resource` 变量在嵌入语句中都是只读的。

下列形式的 `using` 语句

```
using (expression) statement
```

同样具有上述两种可能的扩展，但在这种情况下 `ResourceType` 隐式地为 `expression` 的编译时类型，而 `resource` 变量在嵌入语句中既不可访问，也不可见。

如果资源获取采用局部变量声明的形式，则有可能获取给定类型的多个资源。下列形式的 `using` 语句

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

完全等效于嵌套 `using` 语句的序列：

```
using (ResourceType r1 = e1)
using (ResourceType r2 = e2)
...
using (ResourceType rN = eN)
statement
```

下面的示例创建一个名为 `log.txt` 的文件并将两行文本写入该文件。然后该示例打开这个文件进行读取，并将它所包含的文本行复制到控制台。

```
using System;
using System.IO;
class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
```

```
        w.WriteLine("This is line one");
        w.WriteLine("This is line two");
    }
    using (TextReader r = File.OpenText("log.txt")) {
        string s;
        while ((s = r.ReadLine()) != null) {
            Console.WriteLine(s);
        }
    }
}
```

由于 `TextWriter` 和 `TextReader` 类实现了 `IDisposable` 接口，因此该示例可以使用 `using` 语句以确保所涉及的文件在写入或读取操作后正确关闭。

第9章 命名空间

C# 程序是利用命名空间组织起来的。命名空间既用做程序的“内部”组织系统，也用做“外部”组织系统（一种向其他程序公开自己拥有的程序元素的方法）。

using 指令（§ 9.3）是用来使命名空间用起来更方便。

9.1 编译单元

编译单元定义了源文件的总体结构。编译单元的组成方式如下：先是零个或多个 using 指令，后跟零个或多个全局属性，然后是零个或多个命名空间成员声明。

compilation-unit: (编译单元:)

using-directives_{opt} global-attributes_{opt} namespace-member-declarations_{opt} (using 指令_{可选} 全局属性_{可选} 命名空间成员声明_{可选})

C# 程序由一个或多个编译单元组成，每个编译单元都用一个单独的源文件来保存。编译 C# 程序时，所有这些编译单元一起进行处理。因此，这些编译单元间可以互相依赖，甚至以循环方式互相依赖。

编译单元中的 using 指令的作用范围包括该编译单元内的所有全局属性和命名空间成员声明，但是不会影响其他编译单元。

编译单元的全局属性（§ 17）用于指定目标程序集和模块的属性。程序集和模块充当类型的物理容器。程序集可以包含若干个在物理上分离的模块。

程序中各编译单元中的命名空间成员声明用于为一个称为“全局命名空间”的单个声明空间提供成员。例如：

文件 A.cs:

```
class A {}
```

文件 B.cs:

```
class B {}
```

这两个编译单元是为该全局命名空间提供成员的，在本例中它们分别声明了具有完全限定名 A 和 B 的两个类。由于这两个编译单元为同一声明空间提供成员，因此如果它们分别包含了一个同名成员的声明，将会导致错误。

9.2 命名空间声明

命名空间声明的组成方式如下：先是关键字 namespace，后跟一个命名空间名称和命名空间体，然后加一个分号（可选）。

namespace-declaration: (命名空间声明:)

namespace qualified-identifier namespace-body ;opt (命名空间 限定标识符 命名空间体 ;可选)

qualified-identifier: (限定标识符:)

identifier (标识符)

qualified-identifier . identifier (限定标识符 . 标识符)

namespace-body: (命名空间体:)

{ using-directives_{opt} namespace-member-declarations_{opt} } ({ using 指令_{可选} 命名空间成员声明_{可选} })

命名空间声明可以作为顶级声明出现在编译单元中,或是作为成员声明出现在另一个命名空间声明内。当命名空间声明作为顶级声明出现在编译单元中时,该命名空间成为全局命名空间的一个成员。当命名空间声明出现在另一个命名空间声明内时,内部的命名空间就成为包含着它的外部命名空间的一个成员。无论是何种情况,命名空间的名称在它所属的命名空间内必须是惟一的。

命名空间隐式地为 **public**, 而且在命名空间的声明中不能包含任何访问修饰符。

在命名空间体内,可选用 **using** 指令来导入其他命名空间和类型的名称,这样,就可以直接地而不是通过限定名来引用它们。可选的命名空间成员声明用于为命名空间的声明空间提供成员。请注意,所有的 **using** 指令都必须出现在任何成员声明之前。

命名空间声明中的限定标识符可以是单个标识符,或者是由“.”标记分隔的标识符序列。后一种形式允许程序直接定义一个嵌套命名空间,而不必按词法嵌套若干个命名空间声明。例如:

```
namespace N1.N2
{
    class A {}
    class B {}
}
```

在语义上等效于

```
namespace N1
{
    namespace N2
    {
        class A {}
        class B {}
    }
}
```

命名空间是可扩充的,两个具有相同的完全限定名的命名空间声明为同一声明空间 (§ 3.3) 提供成员。在下面的示例中,前面的两个命名空间声明为同一声明空间提供了成员。

```
namespace N1.N2
{
    class A {}
}
namespace N1.N2
{
    class B {}
}
```

}

在本例中它们分别声明了具有完全限定名 `N1.N2.A` 和 `N1.N2.B` 的两个类。由于两个声明为同一声明空间提供成员，因此如果它们分别包含一个同名成员的声明，将出现错误。

9.3 using 指令

`using` 指令方便了对在其他命名空间中定义的命名空间和类型的使用。`using` 指令仅影响命名空间或类型名称 (§ 3.8) 和简单名称 (§ 7.5.2) 的名称解析过程，与声明不同，`using` 指令不会将新成员添加到与它们所在的编译单元或命名空间相对应的声明空间中。

`using-directives: (using 指令:)`

`using-directive (using 指令)`

`using-directives using-directive (using 指令 using 指令)`

`using-directive: (using 指令:)`

`using-alias-directive (using 别名指令)`

`using-namespace-directive (using 命名空间指令)`

`using` 别名指令 (§ 9.3.1) 用于为一个命名空间或类型启用一个别名。

`using` 命名空间指令 (§ 9.3.2) 用于导入一个命名空间的类型成员。

`using` 指令的范围扩展到直接包含它的编译单元或命名空间体内的所有命名空间成员声明。具体而言，`using` 指令的范围不包括与它对等的 `using` 指令。因此，对等 `using` 指令互不影响，而且按什么顺序编写它们也无关紧要。

9.3.1 using 别名指令

`using` 别名指令用于为一个命名空间或类型指定一个别名（标识符），该别名在直接包含此指令的编译单元或命名空间体内有效。

`using-alias-directive: (using 别名指令:)`

`using identifier = namespace-or-type-name ; (using 标识符 = 命名空间或类型名称 ;)`

在包含 `using` 别名指令的编译单元或命名空间体内的成员声明中，由 `using` 别名指令引入的标识符可用于引用给定的命名空间或类型。例如：

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using A = N1.N2.A;
    class B: A {}
}
```

上面的示例中，在 `N3` 命名空间中声明成员时，`A` 是 `N1.N2.A` 的别名，因此类 `N3.B`

从类 N1.N2.A 派生。通过为 N1.N2 创建别名 R 然后引用 R.A 可以得到同样的效果：

```
namespace N3
{
    using R = N1.N2;
    class B: R.A {}
}
```

using 别名指令中的标识符在直接包含该 using 别名指令的编译单元或命名空间的声明空间内必须是惟一的。例如：

```
namespace N3
{
    class A {}
}
namespace N3
{
    using A = N1.N2.A;    //错误，A 已经存在！
}
```

上例中，N3 已包含了成员 A，因此 using 别名指令使用 A 作为标识符会导致编译时错误。同样，如果同一个编译单元或命名空间体中的两个或更多 using 别名指令用相同名称来声明别名，也会导致编译时错误。

using 别名指令使别名可用在特定编译单元或命名空间体内，但是它不会将任何新成员添加到相应的声明空间。换句话说，using 别名指令是不可传递的，它指定的别名仅在它所在的编译单元或命名空间体内有效。下面的示例：

```
namespace N3
{
    using R = N1.N2;
}
namespace N3
{
    class B: R.A {}    //错误，R 是未知的
}
```

其中，引入 R 的 using 别名指令的范围只延伸到包含它的命名空间体中的成员声明，因此 R 在第二个命名空间声明中是未知的。但是，如果将 using 别名指令放置在包含它的编译单元中，则该别名在两个命名空间声明中都可用：

```
using R = N1.N2;
namespace N3
{
    class B: R.A {}
}
namespace N3
{
    class C: R.A {}
}
```

和常规成员一样，using 别名指令引入的别名在嵌套范围中也可被具有相似名称的成员所隐藏。下面的示例：

```
using R = N1.N2;
namespace N3
```

```

{
    class R {}
    class B: R.A {}      //错误, R 中没有成员 A
}

```

其中, B 的声明中对 R.A 的引用将导致编译时错误, 原因是这里的 R 所引用的是 N3.R 而不是 N1.N2。

编写 using 别名指令的顺序并不重要, 对在 using 别名指令引用的命名空间或类型名称的解析过程既不受 using 别名指令本身影响, 也不受直接包含着该指令的编译单元或命名空间体中的其他 using 指令影响。换句话说, 对 using 别名指令的命名空间或类型名称的解析, 就如同在直接包含该指令的编译单元或命名空间体中根本没有 using 指令一样来处理。在下面的示例中, 最后一个 using 别名指令导致编译时错误, 原因是它不受第一个 using 别名指令的影响。

```

namespace N1.N2 {}
namespace N3
{
    using R1 = N1;      // 没问题
    using R2 = N1.N2;   // 没问题
    using R3 = R1.N2;   // 错误, R1 是未知的
}

```

using 别名指令可以为所有命名空间或类型创建别名, 包括它在其中出现的命名空间本身, 以及嵌套在该命名空间中的其他任何命名空间或类型。

对一个命名空间或类型进行访问时, 无论用它的别名, 还是用它的所声明的名称, 结果是完全相同的。例如:

```

namespace N1.N2
{
    class A {}
}
namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;
    class B
    {
        N1.N2.A a;      // 引用 N1.N2.A
        R1.N2.A b;      // 引用 N1.N2.A
        R2.A c;         // 引用 N1.N2.A
    }
}

```

其中, 名称 N1.N2.A, R1.N2.A 和 R2.A 是等效的, 它们都引用完全限定名为 N1.N2.A 的类。

9.3.2 using 命名空间指令

using 命名空间指令将一个命名空间中所包含的类型导入到直接封闭该指令的编译单元或命名空间体中, 从而可以直接使用这些被导入的类型的标识符而不必加上它们的限定

名。

using-namespace-directive: (using 命名空间指令:)

`using namespace-name;` (using 命名空间名称;)

在包含 **using** 命名空间指令的编译单元或命名空间体中的成员声明内,可以直接引用包含在给定命名空间中的那些类型。例如:

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using N1.N2;
    class B: A {}
}
```

上面的示例中,在 **N3** 命名空间中的成员声明内,**N1.N2** 的类型成员是直接可用的,所以类 **N3.B** 从类 **N1.N2.A** 派生。

using 命名空间指令导入包含在给定命名空间中的类型,但要注意,它不导入嵌套的命名空间。下面是一个示例:

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using N1;
    class B: N2.A {}    // 错误, N2 未知
}
```

其中,**using** 命名空间指令导入包含在 **N1** 中的类型,但是不导入嵌套在 **N1** 中的命名空间。因此,在 **B** 的声明中引用 **N2.A** 将导致编译时错误,原因是在涉及的范围内没有名为 **N2** 的成员。

与 **using** 别名指令不同,**using** 命名空间指令可能导入一些类型,它们的标识符在封闭该指令的编译单元或命名空间体内已被定义。事实上,**using** 命名空间指令导入的名称会被封闭该指令的编译单元或命名空间体中具有类似名称的成员所隐藏。例如:

```
namespace N1.N2
{
    class A {}
    class B {}
}
namespace N3
{
    using N1.N2;
    class A {}
}
```

其中,在 **N3** 命名空间中的成员声明内,**A** 引用 **N3.A** 而不是 **N1.N2.A**。

当由同一编译单元或命名空间体中的 **using** 命名空间指令导入多个命名空间时,如果它们所包含的类型中有重名的,则直接引用该名称就被认为是不明确的。下面的示例:

```

namespace N1
{
    class A {}
}
namespace N2
{
    class A {}
}
namespace N3
{
    using N1;
    using N2;
    class B: A {}           //错误, A 是不明确的
}

```

其中, N1 和 N2 都包含一个成员 A, 而由于 N3 将两者都导入, 所以在 N3 中引用 A 会导致编译时错误。这种名称冲突, 有两种解决办法: 使用限定名来引用 A, 或者利用 using 别名指令为想要引用的某个特定的 A 启用一个别名。例如:

```

namespace N3
{
    using N1;
    using N2;
    using A = N1.A;
    class B: A {}           // A代表 N1.A
}

```

与 using 别名指令一样, using 命名空间指令不会将任何新成员添加到与它所在的编译单元或命名空间相关的声明空间, 因而, 它仅在该编译单元或者命名空间体内有效。

对 using 命名空间指令所引用的命名空间名称的解析过程, 与对 using 别名指令所引用的命名空间或类型名称的解析过程相同。因此, 同一编译单元或命名空间体中的 using 命名空间指令不会互相影响, 而且可以按照任何顺序编写。

9.4 命名空间成员

命名空间成员声明或者是一个命名空间声明 (§ 9.2), 或者是一个类型声明 (§ 9.5)。

namespace-member-declarations: (命名空间成员声明:)

namespace-member-declaration (命名空间成员声明)

namespace-member-declarations namespace-member-declaration (命名空间成员声明 命名空间成员声明)

namespace-member-declaration: (命名空间成员声明:)

namespace-declaration (命名空间声明)

type-declaration (类型声明)

编译单元或命名空间体可以包含命名空间成员声明, 而此类声明则为与包含它们的编译单元或命名空间体相关的声明空间提供新成员。

9.5 类型声明

类型声明是类声明 (§ 10.1)、结构声明 (§ 11.1)、接口声明 (§ 13.1)、枚举声明 (§ 14.1) 或委托声明 (§ 15.1)。

type-declaration: (类型声明:)

class-declaration (类声明)

struct-declaration (结构声明)

interface-declaration (接口声明)

enum-declaration (枚举声明)

delegate-declaration (委托声明)

类型声明可以作为顶级声明出现在编译单元中，或者作为成员声明出现在命名空间、类或结构内部。

当类型 *T* 的类型声明作为编译单元中的顶级声明出现时，新声明的类型的完全限定名正好是 *T*。当类型 *T* 的类型声明出现在命名空间、类或结构内时，新声明的类型的完全限定名是 *N.T*，其中 *N* 是包含它的命名空间、类或结构的完全限定名。

在类或结构内声明的类型称为嵌套类型 (§ 10.2.6)。

在类型声明中允许使用哪些访问修饰符及具有何种默认访问属性，取决于该声明发生处的上下文 (§ 3.5.1)。

- 在编译单元和命名空间中声明的类型可以具有 `public` 或 `internal` 访问属性。默认为 `internal` 访问属性。
- 在类中声明的类型可以具有 `public`, `protected internal`, `protected`, `internal` 或 `private` 访问属性。默认为 `private` 访问属性。
- 在结构中声明的类型可以具有 `public`, `internal` 或 `private` 访问属性。默认为 `private` 访问属性。

第 10 章 类

类是一种数据结构，它可以包含数据成员（常数和字段）、函数成员（方法、属性、事件、索引器、运算符、实例构造函数、静态构造函数和析构函数），以及嵌套类型。类类型支持继承，继承是一种机制，它使派生类可以对基类进行扩展和专用化。

10.1 类声明

类声明是一种类型声明 (§9.5)，它用于声明一个新的类。

class-declaration: (类声明:)

attributes_{opt} class-modifiers_{opt} class identifier class-base_{opt} class-body ;_{opt} (属性_{可选} 类修饰符_{可选} class 标识符 类基_{可选} 类体 ;_{可选})

类声明的组成方式如下：先是一组特性 (§17) (可选)，后跟一组类修饰符 (§10.1.1) (可选)，然后是关键字 `class` 和一个用来命名该类的标识符，接着是一个类基规范 (`class-base`; 参见§10.1.2) (可选)，然后就是类体 (§10.1.3)，最后有可能接一个分号 (可选)。

10.1.1 类修饰符

类声明可以根据需要包括一个类修饰符序列：

class-modifiers: (类修饰符:)

class-modifier (类修饰符)

class-modifiers class-modifier (类修饰符 类修饰符)

class-modifier: (类修饰符:)

`new`

`public`

`protected`

`internal`

`private`

`abstract`

`sealed`

同一个修饰符在一个类声明中多次出现会导致编译时错误。

`new` 修饰符适用于嵌套类。它表示所修饰的类会把继承下来的同名成员隐藏起来，如 §10.2.2 中所描述。如果 `new` 修饰符出现在一个类声明中，而该声明又不是在一个嵌套类声

明，则导致编译时错误。

`public`、`protected`、`internal` 和 `private` 修饰符控制类的可访问性。根据类声明出现处的上下文，这些修饰符中有些可能不允许使用 (§3.5.1)。

以下几节将对 `abstract` 和 `sealed` 修饰符进行讨论。

10.1.1.1 抽象类

`abstract` 修饰符用于表示所修饰的类是不完整的，并且它只能用做基类。抽象类与非抽象类在以下方面是不同的：

- 抽象类不能直接实例化，并且对抽象类使用 `new` 运算符会导致编译时错误。虽然一些变量和值在编译时的类型可以是抽象的，但是这样的变量和值必须或者为 `null`，或者含有对非抽象类的实例的引用（此非抽象类是从抽象类派生的）。
- 允许（但不要求）抽象类包含抽象成员。
- 抽象类不能被密封。

当从抽象类派生非抽象类时，这些非抽象类必须具体实现所继承的所有抽象成员，从而重写那些抽象成员。下面是一个示例：

```
abstract class A
{
    public abstract void F();
}
abstract class B: A
{
    public void G() {}
}
class C: B
{
    public override void F() {
        // F 方法的实际实现
    }
}
```

其中，抽象类 `A` 引入抽象方法 `F`。类 `B` 引入另一个方法 `G`，但由于它不提供 `F` 的实现，因此 `B` 也必须声明为抽象类。类 `C` 重写 `F`，并提供一个具体实现。由于 `C` 中没有了抽象成员，因此可以（但不要求）将 `C` 声明为非抽象类。

10.1.1.2 密封类

`sealed` 修饰符用于防止从所修饰的类派生出其他类。如果一个密封类被指定为其他类的基类，则会发生编译时错误。

密封类不能同时为抽象类。

`sealed` 修饰符主要用于防止非有意的派生，但是它还能促使某些运行时优化。具体而言，由于密封类永远不会有任何派生类，所以对密封类的实例的虚拟函数成员的调用可以转换为非虚拟调用来处理。

10.1.2 类基规范

类声明可以包含类基规范，它定义该类的直接基类和由该类实现的接口 (§13)。

class-base: (类基:)

 : **class-type** (: 类类型)

 : **interface-type-list** (: 接口类型列表)

 : **class-type** , **interface-type-list** (: 类类型 , 接口类型列表)

interface-type-list: (接口类型列表:)

interface-type (接口类型)

interface-type-list , **interface-type** (接口类型列表 , 接口类型)

10.1.2.1 基类

当类类型包含在类基中时，表示该类就是所声明的类的直接基类。如果类声明中没有类基，或所含的类基只列出接口类型，就假定直接基类就是 `object`。类会从它的直接基类继承成员，如§10.2.1 中所描述的那样。

看下面的示例：

```
class A {}
class B: A {}
```

其中，称类 `A` 为类 `B` 的直接基类，而称 `B` 是从 `A` 派生的。由于 `A` 没有显式地指定直接基类，因此它的直接基类隐含地为 `object`。

类类型的直接基类必须至少与该类类型本身具有同样的可访问性 (§3.5.4)。例如，试图从 `private` 或 `internal` 类派生一个 `public` 类，会导致编译时错误。

类类型的直接基类不能是下列类型：`System.Array`，`System.Delegate`，`System.Enum` 或 `System.ValueType`。

类的基类包括它的直接基类及该直接基类的基类。换句话说，基类集是直接基类关系的传递闭包。在上面的例子中，`B` 的基类就包括 `A` 和 `object`。

除了类 `object`，每个类有且只有一个直接基类。`object` 类没有任何直接基类，并且是所有其他类的终极基类。

当类 `B` 从类 `A` 派生时，`A` 依赖于 `B` 是编译时错误。类“直接依赖于”它的直接基类（如果有的话），并且还“直接依赖于”它直接嵌套于其中的类（如果有的话）。从上述定义可以推出：一个类所依赖的类的完备集就是此“直接依赖于”关系的传递闭包。

示例

```
class A: B {}
class B: C {}
class C: A {}
```

是错误的，因为这些类之间循环依赖。同样，示例

```
class A: B.C {}
class B: A
{
```

```
public class C {}
}
```

也会导致编译时错误，原因是 A 依赖于 B.C（它的直接基类），B.C 依赖于 B（它的直接封闭类），而 B 又循环地依赖于 A。

请注意，类不依赖于嵌套在它内部的类。在下面的示例中，

```
class A
{
    class B: A {}
}
```

B 依赖于 A（原因是 A 既是它的直接基类又是它的直接封闭类），但是 A 不依赖于 B（因为 B 既不是 A 的基类也不是 A 的封闭类）。因此，此示例是有效的。

不能从 `sealed` 类派生出别的类。在下面的示例中，类 B 将发生错误，因为它试图从 `sealed` 类 A 派生。

```
sealed class A {}
class B: A {} // 错误，不能从密封类中派生
```

10.1.2.2 接口实现

类基规范中可以包含接口类型列表，这表示所声明的类实现所列出的各个接口类型。§13.4 将对接口实现进行进一步讨论。

10.1.3 类体

类的类体用于定义该类的成员。

class-body:（类体:）

{ class-member-declarations_{opt} }（{ 类成员声明_{可选} }

10.2 类成员

类的成员由两部分组成：由它的类成员声明引入的成员；从它的直接基类继承来的成员。

class-member-declarations:（类成员声明:）

class-member-declaration（类成员声明）

class-member-declarations class-member-declaration（类成员声明 类成员声明）

class-member-declaration:（类成员声明:）

constant-declaration（常数声明）

field-declaration（字段声明）

method-declaration（方法声明）

property-declaration（属性声明）

event-declaration（事件声明）

indexer-declaration (索引器声明)
 operator-declaration (运算符声明)
 constructor-declaration (构造函数声明)
 destructor-declaration (析构函数声明)
 static-constructor-declaration (静态构造函数声明)
 type-declaration (类型声明)

类的成员分为下列几种类别:

- 常数, 表示与该类相关联的常数值 (§10.3)。
- 字段, 即该类的变量 (§10.4)。
- 方法, 用于实现可由该类执行的计算和操作 (§10.5)。
- 属性, 用于定义一些命名特性, 以及与读取和写入这些特征相关的行为 (§10.6)。
- 事件, 用于定义可由该类生成的通知 (§10.7)。
- 索引器, 使该类的实例可按与数组相同的 (语法) 方式进行索引 (§10.8)。
- 运算符, 用于定义表达式运算符, 通过它对该类的实例进行运算 (§10.9)。
- 实例构造函数, 用于规定在初始化该类的实例时需要做些什么 (§10.10)。
- 析构函数, 用于规定在永久地放弃该类的一个实例之前需要做些什么 (§10.12)。
- 静态构造函数, 用于规定在初始化该类自身时需要做些什么 (§10.11)。
- 类型, 用于表示一些类型, 它们是该类的局部类型 (§9.5)。

可以包含可执行代码的成员统称为该类的函数成员。类的函数成员包括: 方法、属性、事件、索引器、运算符、实例构造函数、析构函数和静态构造函数。

类声明将创建新的声明空间 (§3.3), 而直接包含在该类声明内的类成员声明将在此声明空间中引入新成员。下列规则适用于类成员声明:

- 实例构造函数、静态构造函数和析构函数必须具有与直接封闭它们的类相同的名称。所有其他成员的名称必须与该类的名称不同。
- 常数、字段、属性、事件或类型的名称必须不同于在同一个类中声明的所有其他成员的名称。
- 方法的名称必须不同于在同一个类中声明的所有其他非方法的名称。此外, 方法的签名 (§3.6) 必须不同于在同一个类中声明的所有其他方法的签名。
- 实例构造函数的签名必须不同于在同一个类中声明的所有其他实例构造函数的签名。
- 索引器的签名必须不同于在同一个类中声明的所有其他索引器的签名。
- 运算符的签名必须不同于在同一个类中声明的所有其他运算符的签名。

类的继承成员 (§10.2.1) 不是类的声明空间的组成部分。因此, 派生类可以使用与所继承的成员相同的名称或签名来声明自己的新成员 (这同时也隐藏了被继承的同名成员)。

10.2.1 继承

类继承它的直接基类的成员。继承意味着类隐式地把它的直接基类的所有成员当做自己的成员, 但基类的实例构造函数、静态构造函数和析构函数除外。以下是继承的一些重

要性质。

- 继承是可传递的。如果 C 从 B 派生，而 B 从 A 派生，那么 C 就会既继承在 B 中声明的成员，又继承在 A 中声明的成员。
- 派生类扩展它的直接基类。派生类可以向它继承的成员添加新成员，但是它不能移除继承成员的定义。
- 实例构造函数、静态构造函数和析构函数是不可继承的，但所有其他成员是可继承的，无论它们所声明的可访问性 (§3.5) 如何。但是，根据它们所声明的可访问性，有些继承成员在派生类中可能是无法访问的。
- 派生类可以通过声明具有相同名称或签名的新成员来“隐藏” (§3.7.2) 那个被继承的成员。但是，请注意隐藏继承成员并不移除该成员，而只是使被隐藏的成员在派生类中不可直接访问。
- 类的实例含有在该类中及它的所有基类中声明的所有实例字段的集合，并且存在一个从派生类类型到它的任一基类类型的隐式转换 (§6.1.4)。因此，可以将对某个派生类实例的引用视为对它的任一个基类实例的引用。
- 类可以声明虚拟方法、属性和索引器，而派生类可以重写这些函数成员的实现。这使类展示出多态性行为特征，也就是说，同一个函数成员调用所执行的操作可能是不同的，这取决于用来调用该函数成员的实例的运行时类型。

10.2.2 new 修饰符

类成员声明中可以使用与一个被继承的成员相同的名称或签名来声明一个成员。发生这种情况时，就称该派生类成员隐藏了基类成员。隐藏被继承的成员不算是错误，但这会导致编译器发出警告。若要取消此警告，派生类成员的声明中可以包含一个 `new` 修饰符，表示派生成员是有意隐藏基成员的。§3.7.2 中对该主题进行了进一步讨论。

如果在一个不会隐藏继承成员的声明中使用了 `new` 修饰符，就会为此发出警告。通过移除 `new` 修饰符可取消显示此警告。

10.2.3 访问修饰符

类成员声明中可以使用下列 5 种可能的声明可访问性 (§3.5.1) 中的任何一种：`public`，`protected internal`，`protected`，`internal` 或 `private`。除 `protected internal` 组合外，指定多于一个的访问修饰符会导致编译时错误。当类成员声明不包含任何访问修饰符时，假定为 `private`。

10.2.4 构成类型

在成员声明中所使用的类型称为成员的构成类型。可能的构成类型包括常数、字段、属性、事件或索引器类型、方法或运算符的返回类型，以及方法、索引器、运算符和实例构造函数的参数类型。成员的构成类型必须至少具有与该成员本身相同的可访问性 (§3.5.4)。

10.2.5 静态成员和实例成员

类的成员或者是静态成员，或者是实例成员。一般说来，可以这样来理解：静态成员属于类，而实例成员属于对象（类的实例）。

当字段、方法、属性、事件、运算符和构造函数声明中含有 `static` 修饰符时，它声明静态成员。此外，常数或类型声明会隐式地声明静态成员。静态成员具有下列特征：

- 在 E.M 形式的成员访问 (§7.5.4) 中引用静态成员 M 时，E 必须表示含有 M 的那个类型本身。若 E 表示一个实例，则会导致编译时错误。
- 一个静态字段只标识一个存储位置。无论对一个类创建了多少个实例，它的静态字段永远都只有一个副本。
- 静态函数成员（方法、属性、事件、运算符或构造函数）不能作用于具体的实例，在这类函数成员中引用 `this` 会导致编译时错误。

当字段、方法、属性、事件、索引器、构造函数或析构函数的声明中不包含 `static` 修饰符时，它声明实例成员（实例成员有时称为非静态成员）。实例成员具有以下特点：

- 在 E.M 形式的成员访问 (§7.5.4) 中引用实例成员 M 时，E 必须表示某个含有 M 的类型的实例。E 若表示类型本身，则会导致一个编译时错误。
- 类的每个实例分别包含一组该类的所有实例字段。
- 实例函数成员（方法、属性、索引器、实例构造函数或析构函数）作用于类的特定实例，此实例可以用 `this` 访问 (§7.5.7)。

下列示例阐释访问静态和实例成员的规则：

```
class Test
{
    int x;
    static int y;
    void F() {
        x = 1;           // Ok, 相当于 this.x = 1
        y = 1;           // Ok, 相当于 Test.y = 1
    }
    static void G() {
        x = 1;           // 错误, 不能访问 this.x
        y = 1;           // Ok, 相当于 Test.y = 1
    }
    static void Main() {
        Test t = new Test();
        t.x = 1;          // Ok
        t.y = 1;          // 错误, 不能通过实例访问静态成员
        Test.x = 1;       // 错误, 不能通过类型访问实例成员
        Test.y = 1;       // Ok
    }
}
```

F 方法显示，在实例函数成员中，简单名称 (§7.5.2) 既可用于访问实例成员也可用于访问静态成员。G 方法显示，在静态函数成员中，通过简单名称访问实例成员会导致一个编译时错误。Main 方法显示，在成员访问 (§7.5.4) 中，实例成员必须通过实例访问，静态成员必须通过类型访问。

10.2.6 嵌套类型

在类或结构内声明的类型称为**嵌套类型**（**nested-type**）。在编译单元或命名空间内声明的类型称为**非嵌套类型**（**non-nested-type**）。

下面是一个示例：

```
using System;
class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

其中，类 B 是嵌套类型，这是因为它在类 A 内声明。而由于类 A 在编译单元内声明，因此它是非嵌套类型。

10.2.6.1 完全限定名

嵌套类型的完全限定名 (§3.8) 为 S.N，其中 S 是声明了 N 类型的那个类型的完全限定名。

10.2.6.2 声明可访问性

非嵌套类型可以具有 **public** 或 **internal** 声明可访问性，默认的声明可访问性是 **internal**。嵌套类型也可以具有上述两种声明可访问性，外加一种或更多种其他的声明可访问性，具体取决于包含它的那个类型是类还是结构。

- 在类中声明的嵌套类型可以具有 5 种形式的声明可访问性（**public**，**protected**，**internal**，**protected internal** 或 **private**）中的任一种。与其他类成员一样，默认的声明可访问性是 **private**。
- 在结构中声明的嵌套类型可以具有 3 种声明可访问性（**public**，**internal** 或 **private**）中的任一种，而且与其他结构成员一样，默认的声明可访问性为 **private**。

示例

```
public class List
{
    // 私有数据结构
    private class Node
    {
        public object Data;
        public Node Next;
        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }
}
```

```

private Node first = null;
private Node last = null;
// 公有接口
public void AddToFront(object o) {...}
public void AddToBack(object o) {...}
public object RemoveFromFront() {...}
public object RemoveFromBack() {...}
public int Count { get {...} }
}

```

声明了一个私有嵌套类 `Node`。

10.2.6.3 隐藏

嵌套类型可以隐藏 (§3.7) 基成员。对嵌套类型声明允许使用 `new` 修饰符，以便可以明确表示隐藏。示例

```

using System;
class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}
class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}
class Test
{
    static void Main() {
        Derived.M.F();
    }
}

```

显示嵌套类 `M`，它隐藏了在 `Base` 中定义的方法 `M`。

10.2.6.4 this 访问

关于 `this` 访问 (§7.5.7)，嵌套类型和包含它的那个类型并不具有特殊的关系。准确地说，在嵌套类型内，`this` 不能用于引用包含它的那个类型的实例成员。当需要在嵌套类型内部访问包含它的那个类型的实例成员时，通过将代表所需实例的 `this` 作为一个参数传递给该嵌套类型的构造函数，就可以进行所需的访问了。以下示例

```

using System;
class C
{
    int i = 123;
    public void F() {
        Nested n = new Nested(this);
        n.G();
    }
}

```

```

        public class Nested {
            C this_c;
            public Nested(C c) {
                this_c = c;
            }
            public void G() {
                Console.WriteLine(this_c.i);
            }
        }
    }
    class Test {
        static void Main() {
            C c = new C();
            c.F();
        }
    }
}

```

显示了此技巧。C 实例创建了一个 Nested 实例并将代表它自己的 this 传递给 Nested 的构造函数，这样，就可以对 C 的实例成员进行访问了。

10.2.6.5 对包含类型的私有成员和受保护成员的访问

嵌套类型可以访问包含它的那个类型可访问的所有成员，包括该类型自己的具有 **private** 和 **protected** 声明可访问性的成员。以下示例：

```

using System;
class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}
class Test
{
    static void Main() {
        C.Nested.G();
    }
}

```

显示包含有嵌套类 Nested 的类 C。在 Nested 内，方法 G 调用在 C 中定义的静态方法 F，而 F 具有 **private** 声明可访问性。

嵌套类型还可以访问在包含它的那个类型的基类型中定义的受保护成员。看下面的示例：

```

using System;
class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}

```

```

class Derived: Base
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();      // ok
        }
    }
}
class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}

```

其中，嵌套类 `Derived.Nested` 通过对 `Derived` 的实例进行调用，访问在 `Derived` 的基类 `Base` 中定义的受保护方法 `F`。

10.2.7 保留成员名称

为了便于底层的 C# 运行库的实现，对于每个属性、事件或索引器的源成员声明，任何一个实现都必须根据该成员声明的种类、名称和类型保留两个方法签名。如果程序声明一个成员，而该成员的签名与这些保留签名中的某一个匹配，那么，即使所使用的底层运行库的实现并没有使用这些保留签名，仍将导致一个编译时错误。

保留名称不会引入声明，因此它们不参与成员查找。但是，一个声明若具有相关联的保留方法签名，则该方法签名会参与继承 (§10.2.1)，而且可以使用 `new` 修饰符 (§10.2.2) 将它隐藏起来。

保留这些名称有三个目的：

- 使基础的实现可以通过将普通标识符用做一个方法名称，对 C# 语言的功能进行 `get` 或 `set` 访问。
- 使其他语言可以通过将普通标识符用做一个方法名称，对 C# 语言的功能进行 `get` 或 `set` 访问，从而实现交互操作。
- 使保留成员名称的细节在所有的 C# 实现中保持一致，这有助于确保被一个符合本规范的编译器所接受的源程序也可被另一个编译器接受。

析构函数 (§10.12) 的声明也会导致一个签名被保留 (§10.2.7.4)。

10.2.7.1 为属性保留的成员名称

对于类型 `T` 的属性 `P` (§10.6)，保留了下列签名：

```

T get_P();
void set_P(T value);

```

即使该属性是只读或者只写的，这两个签名仍然都被保留。

在下面的示例中，类 `A` 定义了只读属性 `P`，从而保留了 `get_P` 和 `set_P` 方法的签名。

类 B 从 A 派生并隐藏了这两个保留的签名。

```
using System;
class A
{
    public int P {
        get { return 123; }
    }
}
class B: A
{
    new public int get_P() {
        return 456;
    }
    new public void set_P(int value) {
    }
}
class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}
```

此例产生如下输出：

```
123
123
456
```

10.2.7.2 为事件保留的成员名称

对于委托类型 T 的事件 E (§10.7)，保留了下列签名：

```
void add_E(T handler);
void remove_E(T handler);
```

10.2.7.3 为索引器保留的成员名称

对于类型 T 的具有参数列表 L 的索引器 (§10.8)，保留了下列签名：

```
T get_Item(L);
void set_Item(L, T value);
```

即使索引器是只读或者只写的，这两个签名仍然都被保留。

10.2.7.4 析构函数保留的成员名称

对于包含析构函数 (§10.12) 的类，保留了下列签名：

```
void Finalize();
```

10.3 常数

常数 (constant) 是类成员, 它表示一个常数值 (可以在编译时计算的值)。常数声明可引入一个或多个给定类型的常数。

constant-declaration: (常数声明:)

attributes_{opt} constant-modifiers_{opt} const type constant-declarators ; (属性_{可选} 常数修饰符_{可选} const 类型 常数声明符 ;)

constant-modifiers: (常数修饰符:)

constant-modifier (常数修饰符)

constant-modifiers constant-modifier (常数修饰符 常数修饰符)

constant-modifier: (常数修饰符:)

new
public
protected
internal
private

constant-declarators: (常数声明符:)

constant-declarator (常数声明符)

constant-declarators , constant-declarator (常数声明符 , 常数声明符)

constant-declarator: (常数声明符:)

identifier = constant-expression (标识符 = 常数表达式)

常数声明可包含一组特性 (§17)、一个 new 修饰符 (§10.2.2) 和一个由 4 个访问修饰符 (§10.2.3) 构成的有效组合。属性和修饰符适用于所有由该常数声明所声明的成员。虽然常数被认为是静态成员, 但在常数声明中既不要求也不允许使用 static 修饰符。同一个修饰符在一个常数声明中多次出现是错误的。

常数声明的类型用于指定由声明引入的成员的类型。类型后跟一个常数声明符列表, 该列表中的每个声明符引入一个新成员。常数声明符包含一个用于该命名成员的标识符, 后跟一个 “=” 标记, 然后跟一个对该成员赋值的常数表达式 (§7.15)。

常数声明中指定的类型必须是 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、string、枚举类型或引用类型。每个常数表达式所产生的值, 必须是属于目标类型的, 或者可以通过一个隐式转换 (§6.1) 转换为目标类型。

常数的类型必须至少与常数本身具有同样的可访问性 (§3.5.4)。

常数的值从表达式获取, 此表达式使用简单名称 (§7.5.2) 或成员访问 (§7.5.4)。

常数本身可以出现在常数表达式中。因此, 常数可用在任何需要常数表达式的构造中。这样的构造示例包括 case 标签、goto case 语句、enum 成员声明、属性和其他的常数声明。

如 §7.15 中所描述, 常数表达式是在编译时就可以完全计算出来的表达式。由于创建 string 以外的引用类型的非空值的惟一方法是应用 new 运算符, 但常数表达式中不允许

使用 `new` 运算符，因此，除 `string` 以外的引用类型常数的惟一可能的值是 `null`。

如果需要一个具有常数值符号名称，但是该值的类型不允许在常数声明中使用，或在编译时无法由常数表达式计算出该值，则可以改用 `readonly` 字段 (§10.4.2)。

声明了多个常数的一个常数声明等效于具有相同属性、修饰符和类型的多个常数声明（其中每个声明均只声明一个常数）。例如

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

等效于

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

一个常数可以依赖于同一程序内的其他常数，只要这种依赖关系不是循环的。编译器会自动地安排适当的顺序来计算各个常数声明。下面是一个示例：

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}
class B
{
    public const int Z = A.Y + 1;
}
```

其中，编译器首先计算 `A.Y`，然后计算 `B.Z`，最后计算 `A.X`，产生值 10, 11 和 12。常数声明也可以依赖于其他程序中的常数，但这种依赖关系只能是单方向的。上面的示例中，如果 `A` 和 `B` 在不同的程序中声明，`A.X` 可以依赖于 `B.Z`，但是 `B.Z` 就无法同时再依赖于 `A.Y` 了。

10.4 字段

字段（`field`）是一种表示与对象或类关联的变量的成员。字段声明用于引入一个或多个给定类型的字段。

field-declaration:（字段声明：）

attributes_{opt} field-modifiers_{opt} type variable-declarators ;

（属性_{可选} 字段修饰符_{可选} 类型 变量声明符 ；）

field-modifiers:（字段修饰符：）

field-modifier（字段修饰符）

field-modifiers field-modifier（字段修饰符 字段修饰符）

field-modifier: (字段修饰符:)

```
new
public
protected
internal
private
static
readonly
volatile
```

variable-declarators: (变量声明符:)

variable-declarator (变量声明符)

variable-declarators , variable-declarator (变量声明符 , 变量声明符)

variable-declarator: (变量声明符:)

identifier (标识符)

identifier = variable-initializer (标识符 = 变量初始值设定项)

variable-initializer: (变量初始值设定项:)

expression (表达式)

array-initializer (数组初始值设定项)

字段声明可以包含一组特性 (§17)，一个 `new` 修饰符 (§10.2.2)，关于 4 个访问修饰符 (§10.2.3) 的一个有效组合和一个 `static` 修饰符 (§10.4.1)。此外，字段声明还可以包含一个 `readonly` 修饰符 (§10.4.2) 或一个 `volatile` 修饰符 (§10.4.3)，但不能两者都包含。属性和修饰符适用于由该字段声明所声明的所有成员。同一个修饰符在一个字段声明中多次出现是错误的。

字段声明的类型用于指定由该声明引入的成员的类型。类型后跟一个变量声明符列表，其中每个变量声明符引入一个新成员。变量声明符包含一个用于命名该成员的标识符，还可以根据需要再后跟一个“=”标记，以及一个用于赋予成员初始值的变量初始值设定项 (§10.4.5)。

字段的类型必须至少具有与字段本身同样的可访问性 (§3.5.4)。

字段的值从一个表达式获得，该表达式使用简单名称 (§7.5.2) 或成员访问 (§7.5.4)。使用赋值 (§7.13) 修改非只读字段的值。非只读字段的值可以使用后缀增量和减量运算符 (§7.5.9) 及前缀增量和减量运算符 (§7.6.5) 获取和修改。

声明了多个字段的一个字段声明等效于具有相同属性、修饰符和类型的多个字段的声明（其中每个声明均只声明一个字段）。例如

```
class A
{
    public static int X = 1, Y, Z = 100;
}
```

等效于

```
class A
{
```

```
public static int X = 1;
public static int Y;
public static int Z = 100;
```

10.4.1 静态字段和实例字段

当字段声明中含有 `static` 修饰符时，由该声明引入的字段为静态字段（`static field`）。当不存在 `static` 修饰符时，由该声明引入的字段为实例字段（`instance field`）。静态字段和实例字段是 C# 所支持的几种变量（§5）中的两种，它们有时被分别称为静态变量和实例变量。

静态字段不属于某个特定的实例；相反，它只标识了一个存储位置。不管创建了多少个类实例，对于关联的应用程序域来说，在任何时候静态字段都只会有一个副本。

实例字段属于某个实例。具体说来，类的每个实例都包含了该类的所有实例字段的一个单独的集合。

若用 `E.M` 的成员访问形式（§7.5.4）来引用一个字段，如果 `M` 是静态字段，则 `E` 必须表示含有 `M` 的一个类型，但如果 `M` 是实例字段，则 `E` 必须表示一个含有 `M` 的类型的某个实例。

§10.2.5 对静态成员和实例成员之间的差异进行了进一步讨论。

10.4.2 只读字段

当字段声明中含有 `readonly` 修饰符时，该声明所引入的字段为只读字段。给只读字段的直接赋值只能作为声明的组成部分出现，或在同一类中的实例构造函数或静态构造函数中出现（在这些上下文中，只读字段可以被多次赋值）。准确地说，只在下列上下文中允许对只读字段进行直接赋值：

- 在用于引入该字段（通过添加一个变量初始值设定项）的变量声明符中。
- 对于实例字段，在包含字段声明的类的实例构造函数中；对于静态字段，在包含字段声明的类的静态构造函数中。也只有在这些上下文中，将 `readonly` 字段作为 `out` 或 `ref` 参数传递才有效。

在其他任何上下文中，试图对只读字段进行赋值或将它作为 `out` 或 `ref` 参数传递都会导致一个编译时错误。

10.4.2.1 对常数使用静态只读字段

如果需要一个具有常数值符号名称，但该值的类型不允许在 `const` 声明中使用，或者无法在编译时计算出该值，则 `static readonly` 字段就可以发挥作用了。下面是一个示例：

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
}
```

```

    public static readonly Color Blue = new Color(0, 0, 255);
    private byte red, green, blue;
    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}

```

其中, Black, White, Red, Green 和 Blue 成员不能被声明为 `const` 成员, 这是因为在编译时无法计算它们的值。不过, 将它们声明为 `static readonly` 能达到基本相同的效果。

10.4.2.2 常数和静态只读字段的版本控制

常数和只读字段具有不同的二进制版本控制语义。当表达式引用常数时, 该常数的值在编译时获取, 但是当表达式引用只读字段时, 要等到运行时才获取该字段的值。请考虑一个包含两个单独程序的应用程序:

```

using System;
namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}
namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}

```

Program1 和 Program2 命名空间表示两个单独编译的程序。由于 Program1.Utils.X 声明为静态只读字段, 因此 Console.WriteLine 语句要输出的值在编译时是未知的, 在运行时才能获取。这样, 如果更改 X 的值并重新编译 Program1, 则即使 Program2 未被重新编译, Console.WriteLine 语句也将输出新值。但是, 假如 X 是常数, 那么 X 的值将在编译 Program2 时获取, 并且在重新编译 Program2 之前不会受到 Program1 中的更改的影响。

10.4.3 易失字段

当字段声明中含有 `volatile` 修饰符时, 该声明引入的字段为易失字段 (volatile field)。

由于采用了优化技术 (它会重新安排指令的执行顺序), 在多线程的程序运行环境下, 如果不采取同步 (如由 `lock` 语句 (§8.12) 所提供的) 控制手段, 则对于非易失字段的访问可能会导致意外的和不可预见的结果。这些优化可以由编译器、运行时系统或硬件执行。但是, 对于易失字段, 优化时的这种重新排序必须遵循以下规则:

- 读取一个易失字段称为易失读取 (volatile read)。易失读取具有“获取语义”, 也

就是说，按照指令序列，所有排在易失读取之后的对内存的引用，在执行时也一定排在它的后面。

- 写入一个易失字段称为易失写入 (volatile write)。易失写入具有“释放语义”，也就是说，按照指令序列，所有排在易失写入之前的对内存的引用，在执行时也一定排在它的前面。

这些限制能确保所有线程都会观察到由其他任何线程所执行的易失写入（按照原来安排的顺序）。一个遵循本规范的实现并非必须要使易失写入的执行顺序，在所有正在执行的线程看来都是一样的。易失字段的类型必须是下列类型中的一种：

- 引用类型。
- 类型 `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `char`, `float` 或 `bool`。
- 枚举基类型为 `byte`, `sbyte`, `short`, `ushort`, `int` 或 `uint` 的枚举类型。

示例：

```
using System;
using System.Threading;
class Test
{
    public static int result;
    public static volatile bool finished;
    static void Thread2() {
        result = 143;
        finished = true;
    }
    static void Main() {
        finished = false;
        // 在新的线程中运行 Thread2()
        new Thread(new ThreadStart(Thread2)).Start();
        // 等待 Thread2 发出信号，即通过将 finished 设为 true，得到了 result
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

产生下列输出：

```
result = 143
```

在本例中，方法 `Main` 启动一个新线程，该线程运行方法 `Thread2`。该方法将一个值存储在叫做 `result` 的非易失字段中，然后将 `true` 存储在易失字段 `finished` 中。主线程等待字段 `finished` 被设置为 `true`，然后读取字段 `result`。由于 `finished` 已被声明为 `volatile`，因此主线程从字段 `result` 读取的值一定是 143。如果字段 `finished` 未被声明为 `volatile`，则对 `finished` 和 `result` 的写入顺序就可能改变，这样，当主线程读取字段 `result` 时，它可能尚未被赋值（因而为初始值 0）。将 `finished` 声明为 `volatile` 字段可以防止这种不一致性。

10.4.4 字段初始化

字段（无论是静态字段还是实例字段）的初始值都是字段的类型的默认值（§5.2）。在此默认初始化发生之前是不可能看到字段的值的，因此字段永远不会是“未初始化的”。

示例

```
using System;
class Test
{
    static bool b;
    int i;
    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

产生输出：

```
b = False, i = 0
```

这是因为 `b` 和 `i` 都被自动初始化为默认值。

10.4.5 变量初始值设定项

字段声明可以包含变量初始值设定项（`variable-initializer`）。对于静态字段，变量初始值设定项相当于在类初始化期间执行的赋值语句。对于实例字段，变量初始值设定项相当于创建类的实例时执行的赋值语句。

示例

```
using System;
class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";
    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

产生输出：

```
x = 1.4142135623731, i = 100, s = Hello
```

这是因为对 `x` 的赋值发生在静态字段初始值设定项执行时，而对 `i` 和 `s` 的赋值发生在实例字段初始值设定项执行时。

§10.4.4 中描述的默认值初始化对所有字段都发生，包括具有变量初始值设定项的字段。因此，当初始化一个类时，首先将该类中的所有静态字段初始化为它们的默认值，然后以文本顺序执行各个静态字段初始值设定项。类似地，创建类的实例时，首先将该实例

中的所有实例字段初始化为它们的默认值，然后以文本顺序执行各个实例字段初始值设定项。

在具有变量初始值设定项的静态字段处于默认值状态时，也有可能访问它们。但是，为了培养良好的编程风格，强烈建议不要这么做。示例

```
using System;
class Test
{
    static int a = b + 1;
    static int b = a + 1;
    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

显示此行为。尽管 `a` 和 `b` 的定义是循环的，此程序仍是有效的。它产生以下输出：

```
a = 1, b = 2
```

这是因为静态字段 `a` 和 `b` 在它们的初始值设定项执行之前被初始化为 0（`int` 的默认值）。当 `a` 的初始值设定项运行时，`b` 的值为零，所以 `a` 被初始化为 1。当 `b` 的初始值设定项运行时，`a` 的值已经为 1，因此 `b` 被初始化为 2。

10.4.5.1 静态字段初始化

类的静态字段变量初始值设定项相当于一个赋值序列，这些赋值按照它们在相关的类声明中出现的文本顺序执行。如果类中存在静态构造函数 (§10.11)，则静态字段初始值设定项的执行在该静态构造函数即将执行前发生。否则，静态字段初始值设定项在第一次使用该类的静态字段之前先被执行，但实际执行时间依赖于具体的实现。示例：

```
using System;
class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }
    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}
class A
{
    public static int X = Test.F("Init A");
}
class B
{
    public static int Y = Test.F("Init B");
}
```

会产生如下输出：

```
Init A
Init B
1 1
```

或者产生如下输出：

```
Init B
Init A
1 1
```

这是因为 X 的初始值设定项和 Y 的初始值设定项的执行顺序无法预先确定，上述两种顺序都有可能发生。惟一能够确定的是：它们一定会在引用那些字段之前发生。但是，下面的示例：

```
using System;
class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }
    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}
class A
{
    static A() {}
    public static int X = Test.F("Init A");
}
class B
{
    static B() {}
    public static int Y = Test.F("Init B");
}
```

所产生的输出必然是：

```
Init B
Init A
1 1
```

这是因为关于何时执行静态构造函数的规则（将在§10.11 中定义）规定：B 的静态构造函数（以及 B 的静态字段初始值设定项）必须在 A 的静态构造函数和字段初始值设定项之前运行。

10.4.5.2 实例字段初始化

类的实例字段变量初始值设定项相当于一个赋值序列，它在当控制进入该类的任一个实例构造函数（§10.10.1）时立即执行。变量初始值设定项按它们出现在类声明中的文本顺序执行。§10.10 中对类实例的创建和初始化过程进行了进一步描述。

实例字段的变量初始值设定项不能引用正在创建的实例。因此，在变量初始值设定项中引用 this 会导致编译时错误。同样，在变量初始值设定项中通过简单名称引用任何实例成员也会导致编译时错误。下面是一个示例：

```
class A
{
    int x = 1;
    int y = x + 1;    // 错误，原因是它引用了 this 的实例成员
}
```


其中，y 的变量初始值设定项导致编译时错误，原因是它引用了正在创建的实例的成员。

10.5 方法

方法是一种用于实现可以由对象或类执行的计算或操作的成员。方法是使用方法声明来声明的：

method-declaration: (方法声明:)

method-header method-body (方法头 方法体)

method-header: (方法头:)

attributes_{opt} method-modifiers_{opt} return-type member-name (formal-parameter-list_{opt}) (属性_{可选} 方法修饰符_{可选} 返回类型 成员名 (形参表_{可选}))

method-modifiers: (方法修饰符:)

method-modifier (方法修饰符)

method-modifiers method-modifier (方法修饰符 方法修饰符)

method-modifier: (方法修饰符:)

new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

return-type: (返回类型:)

type (类型)

void

member-name: (成员名:)

identifier (标识符)

interface-type . identifier (接口类型 . 标识符)

method-body: (方法体:)

block (块)

;

方法声明中可包含一组特性 (§17) 和关于 4 个访问修饰符 (§10.2.3) 的一个有效组合，还可含有 new (§10.2.2)，static (§10.5.2)，virtual (§10.5.3)，override (§10.5.4)，sealed (§10.5.5)，abstract (§10.5.6)，以及 extern (§10.5.7) 修饰符。

如果以下所有条件都为真，则所述的声明就具有一个有效的修饰符组合：

- 该声明包含一个有效的访问修饰符 (§10.2.3) 组合。
- 该声明中所含的修饰符没有彼此相同的。
- 该声明最多包含下列修饰符中的一个：static, virtual 和 override。
- 该声明最多包含下列修饰符中的一个：new 和 override。
- 如果该声明包含了 abstract 修饰符，则该声明不包含下列任何修饰符：static, virtual, sealed 或 extern。
- 如果声明包含 private 修饰符，则该声明不包含下列任何修饰符：virtual, override 或 abstract。
- 如果声明包含 sealed 修饰符，则该声明还包含 override 修饰符。

方法声明的返回类型用于指定由该方法计算和返回的值的类型。如果方法不返回一个值，则它的返回类型为 void。

成员名用于指定所声明的方法的名称。若所声明的方法不是关于某个显式接口成员的一个实现 (§13.4.1)，则成员名就是一个标识符。而对于一个显式接口成员实现，成员名应由接口类型后跟一个 “.” 和一个标识符组成。

可选的形参表用于指定方法的参数 (§10.5.1)。

返回类型和在方法的形参表中引用的各个类型必须至少具有和方法本身相同的可访问性 (§3.5.4)。

对于 abstract 和 extern 方法，方法体只有一个分号。对于所有其他方法，方法体由一个块组成，该块用于指定在调用方法时要执行哪些语句。

方法的名称和形参表定义了该方法的签名 (§3.6)。准确地说，一个方法的签名由它的名称及它的形参的数目、每个形参的修饰符和类型组成。返回类型不是方法签名的组成部分，形参的名称也不是。

方法的名称必须不同于在同一个类中声明的所有其他非方法的名称。此外，一个方法的签名必须不同于在同一个类中声明的所有其他方法的签名。

10.5.1 方法参数

一个方法的参数（如果有的话）是由该方法的形参表来声明的。

formal-parameter-list: (形参表:)

fixed-parameters (固定参数)

fixed-parameters , parameter-array (固定参数 , 参数数组)

parameter-array (参数数组)

fixed-parameters: (固定参数:)

fixed-parameter (固定参数)

fixed-parameters , fixed-parameter (固定参数 , 固定参数)

fixed-parameter: (固定参数:)

attributes_{opt} parameter-modifier_{opt} type identifier (属性_{可选} 参数修饰符_{可选} 类型 标识符)

parameter-modifier: (参数修饰符:)

ref

out

parameter-array: (参数数组:)

attributes_{opt} params array-type identifier (属性_{可选} params 数组类型 标识符)

形参表包含一个或多个由逗号分隔的参数,其中只有最后一个参数才可以是参数数组。

固定参数包含一组可选的特性 (§17)、一个可选的 ref 或 out 修饰符、一个类型和一个标识符。每个固定参数均声明了参数,指定了该参数的名称及其所属的类型。

参数数组包含一组可选的特性 (§17)、一个 params 修饰符、一个数组类型和一个标识符。参数数组声明单个具有给定名称且属于给定数组类型的参数。参数数组中的数组类型必须是一维数组类型 (§12.1)。在方法调用中,参数数组可以用单个给定数组类型的参数来表示,也可以用零个或多个该数组元素类型的参数来表示。§10.5.1.4 中对参数数组进行了进一步描述。

方法声明为所声明的参数和局部变量创建了单独的声明空间。该方法的形参表和在方法的块中的局部变量声明把它们所声明的名称提供给此声明空间。方法声明空间中的所有名称都必须是惟一的。因此,参数或局部变量的名称与其他参数或局部变量的名称相同会导致编译时错误。

执行方法调用 (§7.5.5.1) 时,创建关于该方法的形参列表和局部变量的一个副本(仅适用于本次调用),而该调用所提供的参数列表则用于把所含的值或变量引用赋给新创建的形参。在方法的块内,形参可以在简单名称表达式 (§7.5.2) 中由它们的标识符引用。

有 4 种形参:

- 值参数,声明时不带任何修饰符。
- 引用参数,用 ref 修饰符声明。
- 输出参数,用 out 修饰符声明。
- 参数数组,用 params 修饰符声明。

如§3.6 中所描述的,ref 和 out 修饰符是方法签名的组成部分,但 params 修饰符不是。

10.5.1.1 值参数

声明时不带修饰符的参数是值参数。一个值参数相当于一个局部变量,只是它的初始值来自该方法调用所提供的相应参数。

当形参是值参数时,方法调用中的对应参数必须是一个表达式,并且它的类型可以隐式转换 (§6.1) 为形参的类型。

允许方法将新值赋给值参数。这样的赋值只影响由该值参数表示的局部存储位置,而不会影响在方法调用时由调用方给出的实参。

10.5.1.2 引用参数

用 ref 修饰符声明的参数是引用参数。与值参数不同,引用参数并不创建新的存储位置。相反,引用参数表示的存储位置恰是在方法调用中作为参数给出的那个变量所表示的

存储位置。

当形参为引用参数时，方法调用中的对应参数必须由关键字 `ref`，后跟一个与形参类型相同的变量引用 (§5.4) 组成。变量在可以作为引用参数传递之前，必须先明确赋值。

在方法内部，引用参数始终被认为是明确赋值的。

示例

```
using System;
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

产生输出

```
i = 2, j = 1
```

对于 `Main` 中的 `Swap` 的调用，`x` 表示 `i` 而 `y` 表示 `j`。因此，该调用具有交换 `i` 和 `j` 的值的 effect。

在采用引用参数的方法中，多个名称可能表示同一存储位置。以下是一个示例：

```
class A
{
    string s;
    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }
    void G() {
        F(ref s, ref s);
    }
}
```

其中，方法 `G` 中对 `F` 的调用分别为 `a` 和 `b` 传递了一个对 `s` 的引用。因此，对于该调用，名称 `s`、`a` 和 `b` 全都引用同一存储位置，并且三个赋值全都修改了同一个实例字段 `s`。

10.5.1.3 输出参数

用 `out` 修饰符声明的参数是输出参数。与引用参数类似，输出参数不创建新的存储位置。输出参数表示的存储位置恰是在该方法调用中作为参数给出的那个变量所表示的存储位置。

当形参为输出参数时，方法调用中的相应参数必须由关键字 `out`，后跟一个与形参类

型相同的变量引用 (§5.4)^[1] 组成。变量在可以作为输出参数传递之前不一定需要明确赋值，但是在将变量作为输出参数传递的调用之后，该变量被认为是明确赋值的。

在方法内部，与局部变量相同，输出参数最初被认为是未赋值的，因而必须在使用它的值之前明确赋值。

在方法返回之前，该方法的每个输出参数都必须明确赋值。

输出参数通常用在需要产生多个返回值的方法中。例如：

```
using System;
class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }
    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

此例产生输出：

```
c:\Windows\System\
hello.txt
```

请注意，`dir` 和 `name` 变量在它们被传递给 `SplitPath` 之前可以是未赋值的，而它们在调用之后就被认为是明确赋值的了。

10.5.1.4 参数数组

用 `params` 修饰符声明的参数是参数数组。如果形参表包括一个参数数组，则该参数数组必须位于该列表的最后而且它必须是一维数组类型。例如，类型 `string[]` 和 `string[][]` 可用做参数数组的类型，但是类型 `string[,]` 不能。不可能将 `params` 修饰符与 `ref` 和 `out` 修饰符组合起来使用。

在方法调用中，允许以下列两种方式之一来为参数数组指定对应的参数：

- 赋予参数数组的参数可以是单个的表达式，它的类型可以隐式转换 (§6.1) 为该参数数组的类型。在此情况下，参数数组的作用与值参数完全一样。
- 或者，此调用可以为参数数组指定零个或多个参数，其中每个参数都是一个表达式，它的类型可隐式转换 (§6.1) 为该参数数组的元素的类型。在此情况下，此调用创建一个长度对应于参数个数、类型与该参数数组的类型相同的一个数组实

^[1] 原书为§5.3.3“确定明确赋值的细则”，而这里应该是§5.4“变量引用”。

例，用给定的参数值初始化该数组实例的元素，并将新创建的数组实例用做实参。

除了允许在调用中使用可变数量的参数，参数数组与同一类型的值参数 (§10.5.1.1) 完全等效。

示例：

```
using System;
class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
    }
    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

产生输出：

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

F 的第一次调用只是将数组 `arr` 作为值参数传递。F 的第二次调用自动创建一个具有给定元素值的四元素 `int[]` 并将该数组实例作为值参数传递。与此类似，F 的第三次调用创建一个零元素的 `int[]` 并将该实例作为值参数传递。第二次和第三次调用完全等效于编写下列代码：

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

执行重载决策时，具有参数数组的方法可能以它的正常形式适用或以它的扩展形式 (§7.4.2.1) 适用。只有在方法的正常形式不适用，并且在同一类型中尚未声明与方法扩展形式具有相同签名的方法时，上述的方法扩展形式才可供选用。

示例：

```
using System;
class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }
    static void F() {
        Console.WriteLine("F()");
    }
    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }
    static void Main() {
        F();
    }
}
```

```

        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}

```

产生输出:

```

F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);

```

在该示例中，在同一个类中，已经声明了两个常规方法，它们的签名与具有参数数组的那个方法的扩展形式相同。因此，在执行重载决策时不考虑这些扩展形式，因而第一次和第三次方法调用将选择常规方法。当在某个类中声明了一个具有参数数组的方法时，同时再声明一些与该方法的扩展形式具有相同的签名的常规方法，这种情况比较常见。这样做可以避免为数组配置内存空间（若调用具有参数数组的方法的扩展形式，则无法避免）。

当参数数组的类型为 `object[]` 时，在方法的正常形式和单个 `object` 参数的扩展形式之间将产生潜在的多义性。产生此多义性的原因是 `object[]` 本身可隐式转换为 `object`。然而，此多义性并不会造成任何问题，这是因为可以在需要时通过插入一个强制转换来解决它。

示例:

```

using System;
class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.Write(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }
    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}

```

产生输出:

```

System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double

```

在 `F` 的第一次和最后一次调用中，`F` 的正常形式是适用的，这是因为存在一个从自变量类型到参数类型的转换（这里其实两者都是 `object[]` 类型）。因此，重载决策选择 `F` 的正常形式，而且将该参数作为常规的值参数传递。在第二次和第三次调用中，`F` 的正常形

式不适用，这是因为不存在从自变量类型到参数类型的转换（类型 `object` 不能隐式转换为类型 `object[]`）。但是，`F` 的扩展形式是适用的，因此重载决策选择它。因此，这两个调用都创建了一个具有单个元素的、类型为 `object[]` 的数组，并且用给定的参数值（它本身是对一个 `object[]` 的引用）初始化该数组的惟一元素。

10.5.2 静态方法和实例方法

若一个方法声明中含有 `static` 修饰符，则称该方法为静态方法。若其中没有 `static` 修饰符，则称该方法为实例方法。

静态方法不对特定实例进行操作，在静态方法中引用 `this` 会导致编译时错误。

实例方法对类的某个给定的实例进行操作，而且可以用 `this` (§7.5.7) 来访问该实例。

在 `E.M` 形式的成员访问 (§7.5.4) 中引用一个方法时，如果 `M` 是静态方法，则 `E` 必须表示含有 `M` 的一个类型，而如果 `M` 是实例方法，则 `E` 必须表示含有 `M` 的类型的实例。

§10.2.5 对静态成员和实例成员之间的差异进行了更深入的讨论。

10.5.3 虚拟方法

若一个实例方法的声明中包含 `virtual` 修饰符，则称该方法为虚拟方法。若其中没有 `virtual` 修饰符，则称该方法为非虚拟方法。

非虚拟方法的实现是不变的：无论是在声明它的类的实例上还是在派生类的实例上调用该方法，实现都是相同的。与此相反，虚拟方法的实现可以由派生类取代。取代所继承的虚拟方法的实现的过程称为重写该方法 (§10.5.4)。

在虚拟方法调用中，该调用所涉及的那个实例的运行时类型确定了要被调用的究竟是该方法的哪一个实现。在非虚拟方法调用中，相关的实例的编译时类型是决定性因素。准确地说，当在具有编译时类型 `C` 和运行时类型 `R` 的实例（其中 `R` 为 `C` 或者从 `C` 派生的类）上用参数列表 `A` 调用名为 `N` 的方法时，调用按下述规则处理：

- 首先，将重载决策应用于 `C`、`N` 和 `A`，以从在 `C` 中声明的和由 `C` 继承的方法集中选择一个特定的方法 `M`。§7.5.5.1 对此进行了描述。
- 然后，如果 `M` 为非虚拟方法，则调用 `M`。
- 否则（`M` 为虚拟方法），就会调用就 `R` 而言 `M` 的派生程度最大的那个实现。

对于在一个类中声明或者由类继承的每一个虚拟方法，存在一个就该类而言派生程度最大的实现。就类 `R` 而言，虚拟方法 `M` 的派生程度最大的实现按下述规则确定：

- 如果 `R` 中含有关于 `M` 的 `virtual` 声明，则这是 `M` 的派生程度最大的实现。
- 如果 `R` 中含有关于 `M` 的 `override` 声明，则这是 `M` 的派生程度最大的实现。
- 否则，就 `R` 而言 `M` 的派生程度最大的实现与就 `R` 的直接基类而言 `M` 的派生程度最大的实现相同。

下列实例阐释虚拟方法和非虚拟方法之间的区别：

```
using System;
class A
{
    public void F() { Console.WriteLine("A.F"); }
    public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
    new public void F() { Console.WriteLine("B.F"); }
    public override void G() { Console.WriteLine("B.G"); }
}
class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}
```

在该示例中，A 引入一个非虚拟方法 F 和一个虚拟方法 G。类 B 引入一个新的非虚拟方法 F，从而隐藏了继承的 F，并且还重写了继承的方法 G。此例产生输出：

```
A.F
B.F
B.G
B.G
```

请注意，语句 `a.G()` 实际调用的是 `B.G` 而不是 `A.G`。这是因为，对调用哪个实际方法实现起决定作用的是该实例的运行时类型（`B`），而不是该实例的编译时类型（`A`）。

由于一个类中声明的方法可以隐藏继承来的方法，因此同一个类中可以包含若干个具有相同签名的虚拟方法。这不会造成多义性问题，因为除派生程度最大的那个方法外，其他方法都被隐藏起来了。对于以下示例：

```
using System;
class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}
class Test
{
    static void Main() {
```

```

        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}

```

其中, C 类和 D 类均含有两个具有相同签名的虚拟方法: A 引入的虚拟方法和 C 引入的虚拟方法。但是, 由 C 引入的方法隐藏了从 A 继承的方法。因此, D 中的重写声明所重写的是由 C 引入的方法, D 不可能重写由 A 引入的方法。此例产生输出:

```

B.F
B.F
D.F
D.F

```

请注意, 通过访问 D 的实例 (借助一个派生程度较小的类型, 它的方法没有被隐藏起来), 可以调用被隐藏的虚拟方法。

10.5.4 重写方法

若一个实例方法声明中含有 `override` 修饰符, 则称该方法为重写方法 (`override method`)。重写方法用相同的签名重写所继承的虚拟方法。虚拟方法声明用于引入新方法, 而重写方法声明则用于使现有的继承虚拟方法专用化 (通过提供该方法的新实现)。

由 `override` 声明所重写的那个方法称为已重写了的基方法 (`overridden base method`)。对于在类 C 中声明的重写方法 M, 已重写了的基方法是通过检查 C 的各个基类确定的, 检查的方法是: 先从 C 的直接基类开始, 逐个检查每个后续的直接基类, 直至找到一个与 M 具有相同签名的可访问方法。为了查找已重写了的基方法, 可访问方法可以这样来定义: 如果一个方法是 `public`, `protected`, `protected internal` 或者 `internal` 并且与 C 声明在同一程序中, 则认为它是可访问的。

除非下列所有项对于一个重写声明皆为真, 否则会出现编译时错误:

- 可以按照上面描述的规则找到一个已重写了的基方法。
- 该已重写了的基方法是一个虚拟、抽象或重写方法。换句话说, 已重写了的基方法不能是静态或非虚拟方法。
- 已重写了的基方法不是密封方法。
- 重写声明和已重写了的基方法具有相同的返回类型。
- 重写声明和已重写了的基方法具有相同的声明可访问性。换句话说, 重写声明不能更改所对应的虚拟方法的访问性。

重写声明可以使用基访问 (§7.5.8) 访问已重写了的基方法。对于下面的示例:

```

class A
{
    int x;
}

```

```

        public virtual void PrintFields() {
            Console.WriteLine("x = {0}", x);
        }
    }
    class B: A
    {
        int y;
        public override void PrintFields() {
            base.PrintFields();
            Console.WriteLine("y = {0}", y);
        }
    }

```

B 中的调用 `base.PrintFields()` 就调用了在 A 中声明的 `PrintFields` 方法。基访问禁用了虚拟调用机制，而只是简单地将那个重写了的基方法视为非虚拟方法。如果把 B 中的调用改写成 `((A)this).PrintFields()`，它将递归调用在 B 中声明而不是在 A 中声明的 `PrintFields` 方法，这是因为 `PrintFields` 是虚拟的，而且 `((A)this)` 的运行时类型为 B。

只有在包含了 `override` 修饰符时，一个方法才能重写另一个方法。在所有其他情况下，声明一个与继承了的方法具有相同签名的方法只会使那个被继承的方法隐藏起来。对于下面的示例：

```

class A
{
    public virtual void F() {}
}
class B: A
{
    public virtual void F() {}    // 警告，隐藏被继承的 F()
}

```

B 中的 F 方法不包含 `override` 修饰符，因此不重写 A 中的 F 方法。相反，B 中的 F 方法隐藏 A 中的方法，并且由于该声明中没有包含 `new` 修饰符，从而会导致一个警告。

对于下面的示例：

```

class A
{
    public virtual void F() {}
}
class B: A
{
    new private void F() {}    // 在 B 的内部隐藏 A.F
}
class C: B
{
    public override void F() {}    // Ok, 重写 A.F
}

```

B 中的 F 方法隐藏从 A 继承的虚拟 F 方法。因为 B 中的新 F 具有私有访问权限，它的范围只包括 B 的类体而没有延伸到 C。所以，允许 C 中的 F 声明重写从 A 继承的 F。

10.5.5 密封方法

当实例方法声明包含 `sealed` 修饰符时，称该方法为密封方法（sealed method）。如果实例方法声明包含 `sealed` 修饰符，则它必须也包含 `override` 修饰符。使用 `sealed` 修饰符可以防止派生类进一步重写该方法。

对于示例：

```
using System;
class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
    public virtual void G() {
        Console.WriteLine("A.G");
    }
}
class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }
    override public void G() {
        Console.WriteLine("B.G");
    }
}
class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

类 B 提供两个重写方法：具有 `sealed` 修饰符的 F 方法和不具有此修饰符的 G 方法。通过使用 `sealed` 修饰符，B 就可以防止 C 进一步重写 F。

10.5.6 抽象方法

当实例方法声明包含 `abstract` 修饰符时，称该方法为抽象方法（abstract method）。虽然抽象方法同时隐含为虚拟方法，但是它不能有修饰符 `virtual`。

抽象方法声明引入一个新的虚拟方法，但不提供该方法的实现。相反，非抽象的派生类需要重写该方法以提供它们自己的实现。由于抽象方法不提供任何实际实现，因此抽象方法的方法体只包含一个分号。

只允许在抽象类（§10.1.1.1）中使用抽象方法声明。

对于下面的示例：

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}
```

```

public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}
public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}

```

Shape 类定义了一个可以绘制自身的几何形状对象的抽象概念。**Paint** 方法是抽象的，这是因为没有有意义的默认实现。**Ellipse** 类和 **Box** 类是具体的 **Shape** 实现。由于这些类是非抽象的，因此要求它们重写 **Paint** 方法并提供实际实现。

若一个基访问 (§7.5.8) 引用的是一个抽象方法，则会导致一个编译时错误。对于下面的示例：

```

abstract class A
{
    public abstract void F();
}
class B: A
{
    public override void F() {
        base.F();          // 错误, base.F 是抽象的
    }
}

```

调用 **base.F()** 导致了编译时错误，原因是它引用了抽象方法。

在抽象方法声明中可以重写虚拟方法。这使一个抽象类可以强制在它的派生类中重新实现该方法，并使该方法的原始实现不再可用。对于下面的示例：

```

using System;
class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}
abstract class B: A
{
    public abstract override void F();
}
class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}

```

类 **A** 声明一个虚拟方法，类 **B** 用一个抽象方法重写此方法，而类 **C** 重写该抽象方法以提供它自己的实现。

10.5.7 外部方法

当方法声明包含 `extern` 修饰符时，称该方法为外部方法（external method）。外部方法是在外部实现的，通常使用 C# 以外的编程语言。由于外部方法声明不提供任何实际实现，因此外部方法的方法体只包含一个分号。

`extern` 修饰符通常与 `DllImport` 特性（§17.5.1）一起使用，从而使外部方法可以由 DLL（动态链接库）实现。执行环境可以支持其他用来提供外部方法实现的机制。

当外部方法包含 `DllImport` 特性时，该方法声明必须同时包含一个 `static` 修饰符。下面的示例说明 `extern` 修饰符和 `DllImport` 特性的使用：

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;
class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);
    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);
    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);
    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

10.5.8 方法体

方法声明中的方法体或者包含一个块，或者包含一个分号。

抽象方法和外部方法声明不提供方法实现，所以它们的方法体只包含一个分号。对于任何其他方法，方法体是一个块（§8.2），它包含了在调用该方法时应执行的语句。

当方法的返回类型为 `void` 时，不允许该方法体中的 `return` 语句（§8.9.4）指定表达式。如果一个 `void` 方法的方法体的执行正常完成（即控制自方法体的结尾离开），则该方法只是返回到它的调用方。

当方法的返回类型不是 `void` 时，该方法体中的每个 `return` 语句都必须指定一个可隐式转换为返回类型的类型的表达式。对于一个返回值的方法，其方法体的结束点必须是不可到达的。换句话说，在执行返回值的方法中，不允许控制自方法体的结尾离开。

对于下面的示例：

```
class A
{
    public int F() {}           // 错误，返回所需要的值
    public int G() {
        return 1;
    }
    public int H(bool b) {
        if (b) {
            return 1;
        }
    }
}
```



```
        else {  
            return 0;  
        }  
    }  
}
```

返回值的 F 方法导致编译时错误，原因是控制可以超出方法体的结尾。G 方法和 H 方法是正确的，这是因为所有可能的执行路径都以一个指定返回值的 `return` 语句结束。

10.5.9 方法重载

§7.4.2 对方法重载决策规则进行了描述。

10.6 属性

属性（property）是一种用于访问对象或类的特性的成员。属性的示例包括字符串的长度、字体的大小、窗口的标题、客户的名称，等等。属性是字段的自然扩展，这两者都是具有关联类型的命名成员，而且访问字段和属性的语法是相同的。然而，与字段不同，属性不表示存储位置。相反，属性有访问器，这些访问器指定在它们的值被读取或写入时需执行的语句。因此属性提供了一种机制，它把读取和写入对象的某些特性与一些操作关联起来；它们甚至还可以对此类特性进行计算。

属性是使用属性声明声明的：

property-declaration:（属性声明：）

```
attributesopt property-modifiersopt type member-name { accessor-declarations }  
（属性可选 属性修饰符可选 类型 成员名 { 访问器声明 }）
```

property-modifiers:（属性修饰符：）

property-modifier（属性修饰符）

property-modifiers **property-modifier**（属性修饰符 属性修饰符）

property-modifier:（属性修饰符：）

```
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern
```

member-name:（成员名：）

identifier（标识符）

interface-type . identifier (接口类型 . 标识符)

属性声明可包含一组特性 (§17) 和 4 个访问修饰符 (§10.2.3) 的有效组合, 还可包含 new (§10.2.2), static (§10.5.2), virtual (§10.5.3), override (§10.5.4), sealed (§10.5.5), abstract (§10.5.6) 以及 extern (§10.5.7) 修饰符。

对于有效的修饰符组合, 属性声明与方法声明 (§10.5) 遵循相同的规则。

属性声明中的类型用于指定该声明所引入的属性的类型, 而成员名则指定该属性的名称。除非该属性是一个显式的接口成员实现, 否则成员名就只是一个标识符。对于显式接口成员实现 (§13.4.1), 成员名由一个接口类型, 后跟一个 “.” 和一个标识符组成。

属性的类型必须至少与属性本身具有同样的可访问性 (§3.5.4)。

访问器声明 (必须括在 “{ }” 标记中) 声明属性的访问器 (§10.6.2)。访问器指定与属性的读取和写入相关联的可执行语句。

虽然访问属性的语法与访问字段的语法相同, 但是属性并不归类为变量。因此, 不能将属性作为 ref 或 out 参数传递。

属性声明包含 extern 修饰符时, 称该属性为外部属性。因为外部属性声明不提供任何实际的实现, 所以它的每个访问器声明都仅含有一个分号。

10.6.1 静态属性和实例属性

当属性声明包含 static 修饰符时, 称该属性为静态属性 (static property)。当不存在 static 修饰符时, 称该属性为实例属性 (instance property)。

静态属性不与特定实例相关联, 因此在静态属性的访问器内引用 this 会导致编译时错误。

实例属性与类的一个给定实例相关联, 并且该实例可以在属性的访问器内用 this (§7.5.7) 来访问。

在 E.M 形式的成员访问 (§7.5.4) 中引用属性时, 如果 M 是静态属性, 则 E 必须表示包含 M 的类型, 如果 M 是实例属性, 则 E 必须表示包含 M 的类型的一个实例。

§10.2.5 对静态成员和实例成员之间的差异进行了更深入的讨论。

10.6.2 访问器

属性的访问器声明指定与读取和写入该属性相关联的可执行语句。

accessor-declarations: (访问器声明:)

get-accessor-declaration set-accessor-declaration_{opt} (get 访问器声明 set 访问器声明_{可选})

set-accessor-declaration get-accessor-declaration_{opt} (set 访问器声明 get 访问器声明_{可选})

get-accessor-declaration: (get 访问器声明:)

attributes_{opt} get accessor-body (属性_{可选} get 访问器体)

set-accessor-declaration: (set 访问器声明:)

```

    attributesopt set    accessor-body (属性可选 set 访问器体)
accessor-body: (访问器体:)
    block (块)
;

```

访问器声明由一个 `get` 访问器声明或一个 `set` 访问器声明，或者两者一起组成。每个访问器声明都包含标记 `get` 或 `set`，后跟一个访问器体。对于 `abstract` 属性和 `extern` 属性，每个指定访问器的访问器体只是一个分号。对于所有非抽象、非外部属性的访问器，访问器体是一个块，它指定调用相应的访问器时需执行的语句。

`get` 访问器相当于一个具有属性类型返回值的无参数方法。除了作为赋值的目标，当在表达式中引用属性时，将调用该属性的 `get` 访问器以计算该属性的值 (§7.1)。访问器的访问器体必须遵循 §10.5.8 中所描述的适用于返回值方法的规则。具体说来，`get` 访问器的访问器体中的所有 `return` 语句都必须指定一个可隐式转换为属性类型的表达式。另外，`get` 访问器的结束点必须是不可到达的。

`set` 访问器相当于一个具有单个属性类型值参数和 `void` 返回类型的方法。`set` 访问器的隐式参数始终命名为 `value`。当一个属性作为赋值 (§7.13) 的目标，或者作为 `++` 或 `--` 运算符 (§7.5.9, §7.6.5) 的操作数被引用时，就会调用 `set` 访问器，所传递的参数（其值为赋值右边的值或者 `++` 或 `--` 运算符的操作数）将提供新值 (§7.13.1)。访问器的访问器体必须遵循 §10.5.8 所描述的适用于 `void` 方法的规则。具体说来，不允许 `set` 访问器的访问器体中的 `return` 语句指定表达式。由于 `set` 访问器隐式具有名为 `value` 的参数，因此在 `set` 访问器中，若在局部变量或常数声明中出现该名称，则会导致一个编译时错误。

根据 `get` 访问器和 `set` 访问器是否存在，属性可按下列规则分类：

- 同时包含 `get` 访问器和 `set` 访问器的属性称为读写属性。
- 只具有 `get` 访问器的属性称为只读属性。将只读属性作为赋值目标会导致编译时错误。
- 只具有 `set` 访问器的属性称为只写属性。除非是作为赋值的目标，否则在表达式中引用只写属性会导致编译时错误。

对于下面的示例：

```

public class Button: Control
{
    private string caption;
    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }
    public override void Paint(Graphics g, Rectangle r) {
        // 这里是具体的绘图代码
    }
}

```

Button 控件声明了一个公共 Caption 属性。Caption 属性的 get 访问器返回存储在私有 caption 字段中的字符串。set 访问器检查新值是否与当前值不同，如果新值与当前值不同，它将存储新值并重新绘制控件。属性经常遵循上述的模式：get 访问器只返回一个存储在私有字段中的值，而 set 访问器则用于修改该私有字段，然后再执行一些必要的操作，以完全更新所涉及的对象的状态。

使用上边给定的 Button 类，下面是一个使用 Caption 属性的示例：

```
Button okButton = new Button();
okButton.Caption = "OK";           // 调用 set 访问器
string s = okButton.Caption;       // 调用 get 访问器
```

此处，通过给属性赋值调用 set 访问器，而对 get 访问器则通过在表达式中引用该属性来调用。

属性的 get 访问器和 set 访问器都不是独立的成员，也不能单独地声明一个属性的访问器。因此，读写属性的两个访问器不可能具有不同的可访问性。示例

```
class A
{
    private string name;
    public string Name {           // 错误，成员名称累同
        get { return name; }
    }
    public string Name {           // 错误，成员名称累同
        set { name = value; }
    }
}
```

并不是声明单个读写属性。相反，它声明了两个同名的属性，一个是只读的，一个是只写的。由于在同一个类中声明的两个成员不能同名，因此此示例将导致发生一个编译时错误。

当在一个派生类中用与某个所继承的属性相同的名称声明一个新属性时，该派生属性将会隐藏所继承的属性（同时在读取和写入方面）。对于下面的示例：

```
class A
{
    public int P {
        set {...}
    }
}
class B: A
{
    new public int P {
        get {...}
    }
}
```

B 中的 P 属性同时在读取和写入方面隐藏 A 中的 P 属性。因此，对于下列语句：

```
B b = new B();
b.P = 1;           // 错误，B.P 是只读的
((A)b).P = 1;      // Ok，对 A.P 的引用
```

向 b.P 赋值将导致编译时错误，原因是 B 中的只读属性 P 隐藏了 A 中的只写属性 P。

但是，仍可以使用强制转换来访问那个被隐藏了的 P 属性。

与公共字段不同，属性在对象的内部状态和它的公共接口之间提供了一种隔离手段。请看此示例：

```
class Label
{
    private int x, y;
    private string caption;
    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }
    public int X {
        get { return x; }
    }
    public int Y {
        get { return y; }
    }
    public Point Location {
        get { return new Point(x, y); }
    }
    public string Caption {
        get { return caption; }
    }
}
```

其中，Label 类使用两个 int 字段 x 和 y 存储它的位置。该位置同时采用两种方式公开：X 和 Y 属性，以及 Point 类型的 Location 属性。如果在 Label 的未来版本中采用 Point 结构在内部存储此位置更为方便，则可以不影响类的公共接口就完成更改：

```
class Label
{
    private Point location;
    private string caption;
    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }
    public int X {
        get { return location.x; }
    }
    public int Y {
        get { return location.y; }
    }
    public Point Location {
        get { return location; }
    }
    public string Caption {
        get { return caption; }
    }
}
```

相反，如果 x 和 y 是 public readonly 字段，则对 Label 类进行上述更改是不可能的。

通过属性公开状态并不一定比直接公开字段效率低。具体说来，当属性是非虚拟的且只包含少量代码时，执行环境可能会用访问器的实际代码替换对访问器进行的调用。此过

程称为内联 (inlining)，它使属性访问与字段访问一样高效，而且仍保留了属性的更高灵活性。

由于调用 `get` 访问器在概念上等效于读取字段的值，因此使 `get` 访问器具有可见的副作用被认为是不好的编程风格。对于下面的示例：

```
class Counter
{
    private int next;
    public int Next {
        get { return next++; }
    }
}
```

`Next` 属性的值取决于该属性以前被访问的次数。因此，访问此属性会产生可见的副作用，而此属性应当作为一个方法实现。

`get` 访问器的“无副作用”约定并不意味着 `get` 访问器应当始终被编写为只返回存储在字段中的值。事实上，`get` 访问器经常通过访问多个字段或调用方法以计算属性的值。但是，正确设计的 `get` 访问器不会执行任何导致对象的状态发生可见变化的操作。

属性还可用于将某个资源的初始化延迟到第一次引用该资源时执行。例如：

```
using System.IO;
public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;
    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }
    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(Console.OpenStandardOutput());
            }
            return writer;
        }
    }
    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(Console.OpenStandardError());
            }
            return error;
        }
    }
}
```

`Console` 类包含三个属性：`In`、`Out` 和 `Error`，它们分别表示三种标准的设备：输入、输出和错误信息报告。通过将这些成员作为属性公开，`Console` 类可以将它们的初始化延迟到它们被实际使用时。例如，对下列示例：

```
Console.Out.WriteLine("hello, world");
```

仅在第一次引用 Out 属性时，才创建输出设备的基础 TextWriter。但是，如果应用程序不引用 In 和 Error 属性，则不会创建这些设备的任何对象。

10.6.3 虚拟、密封、重写和抽象访问器

virtual 属性声明指定属性的访问器是虚拟的。virtual 修饰符适用于读写属性的两个访问器（读写属性的访问器不可能只有一个是虚拟的）。

abstract 属性声明指定属性的访问器是虚拟的，而且不提供访问器的实际实现。另外，非抽象派生类还要求通过重写属性以提供它们自己的访问器实现。由于抽象属性声明的访问器不提供实际实现，因此它的访问器体只包含一个分号。

同时包含 abstract 修饰符和 override 修饰符的属性声明表示属性是抽象的并且重写一个基属性。此类属性的访问器也是抽象的。

只能在抽象类 (§10.1.1.1) 中使用抽象属性声明。通过用一个含有 override 指令的属性声明，可以在派生类中重写被继承的虚拟属性的访问器，这称为重写属性声明。重写属性声明并不声明新属性。相反，它只是对现有虚拟属性的访问器的实现进行专用化。

重写属性声明必须指定与被继承的属性完全相同的可访问性修饰符、类型和名称。如果被继承的属性只有单个访问器（即该属性是只读或只写的），则重写属性必须只包含该访问器。如果被继承的属性同时包含两个访问器（即该属性是读写的），则重写属性既可以仅包含其中任一个访问器，也可以同时包含两个访问器。

重写属性声明可以包含 sealed 修饰符。使用此修饰符可以防止派生类进一步重写该属性。密封属性的访问器也是密封的。

除了在声明和调用语法中的差异，虚拟、密封、重写和抽象访问器与虚拟、密封、重写和抽象方法具有完全相同的行为。准确地说，§10.5.3，§10.5.4，§10.5.5 和 §10.5.6 中描述的规则都适用，就好像访问器是相应形式的方法一样。

- get 访问器相当于一个无参数方法，该方法具有属性类型的返回值及与包含属性相同的修饰符。
- set 访问器相当于一个方法，该方法具有单个属性类型的值参数、void 返回类型，以及与包含属性相同的修饰符。

对于下面的示例：

```
abstract class A
{
    int y;
    public virtual int X {
        get { return 0; }
    }
    public virtual int Y {
        get { return y; }
        set { y = value; }
    }
    public abstract int Z { get; set; }
}
```


X 是虚拟只读属性，Y 是虚拟读写属性，而 Z 是抽象读写属性。由于 Z 是抽象的，所以包含它的类 A 也必须被声明为抽象的。

下面显示了一个从 A 派生的类：

```
class B: A
{
    int z;
    public override int X {
        get { return base.X + 1; }
    }
    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }
    public override int Z {
        get { return z; }
        set { z = value; }
    }
}
```

此处，X、Y 和 Z 的声明是重写属性声明。每个属性声明都与它们所继承的属性的可访问性修饰符、类型和名称完全匹配。X 的 get 访问器和 Y 的 set 访问器使用 base 关键字来访问所继承的访问器。Z 的声明重写了两个抽象访问器，因此在 B 中不再有抽象的函数成员，B 也可以是非抽象类。

10.7 事件

事件（event）是一种使对象或类能够提供通知的成员。客户端可以通过提供事件处理程序为相应的事件添加可执行代码。

事件是使用事件声明来声明的：

event-declaration: (事件声明:)

attributes_{opt} event-modifiers_{opt} event type variable-declarators ; (属性_{可选} 事件修饰符_{可选} event 类型 变量声明符 ;)

attributes_{opt} event-modifiers_{opt} event type member-name { event-accessor declarations } (属性_{可选} 事件修饰符_{可选} event 类型 成员名 { 事件访问器声明 })

event-modifiers: (事件修饰符:)

event-modifier (事件修饰符)

event-modifiers event-modifier (事件修饰符 事件修饰符)

event-modifier: (事件修饰符:)

```
new
public
protected
internal
private
static
virtual
sealed
```

```

    override
    abstract
    extern

```

event-accessor-declarations: (事件访问器声明:)

add-accessor-declaration remove-accessor-declaration (添加访问器声明 移除访问器声明)

remove-accessor-declaration add-accessor-declaration (移除访问器声明 添加访问器声明)

add-accessor-declaration: (添加访问器声明:)

attributes_{opt} add block (属性_{可选} add 块)

remove-accessor-declaration: (移除访问器声明:)

attributes_{opt} remove block (属性_{可选} remove 块)

事件声明可包含一组特性 (§17) 和 4 个访问修饰符 (§10.2.3) 的有效组合、new (§10.2.2)、static (§10.5.2)、virtual (§10.5.3)、override (§10.5.4)、sealed (§10.5.5)、abstract (§10.5.6) 以及 extern (§10.5.7) 修饰符。

关于有效的修饰符组合，事件声明与方法声明 (§10.5) 遵循相同的规则。

事件声明的类型必须是委托类型 (§4.2)，而该委托类型必须至少具有与事件本身一样的可访问性 (§3.5.4)。

事件声明中可以包含事件访问器声明。但是，如果声明中没有包含事件访问器声明，对于非外部、非抽象事件，编译器将自动提供 (§10.7.1)；对于外部事件，访问器由外部提供。

省略了事件访问器声明的事件声明用于定义一个或多个事件（每个变量声明符各表示一个事件）。事件声明中的属性和修饰符适用于所有由该事件声明所声明的成员。

若事件声明既包含 abstract 修饰符又包含以括号分隔的事件访问器声明，则会导致编译时错误。

当事件声明包含 extern 修饰符时，称该事件为外部事件 (external event)。因为外部事件声明不提供任何实际的实现，所以在一个外部事件声明中既包含 extern 修饰符又包含事件访问器声明是错误的。

事件可用做 += 和 -= 运算符 (§7.13.3) 左边的操作数。这些运算符分别用于将事件处理程序添加到所涉及的事件或从该事件中移除事件处理程序，而该事件的访问修饰符用于控制允许这类运算的上下文。

由于 += 和 -= 是仅有的能够在声明了某个事件的类型的外部对该事件进行的操作，因此，外部代码可以为一个事件添加和移除处理程序，但是不能以其他任何方式来获取或修改基础的事件处理程序列表。

在 $x += y$ 或 $x -= y$ 形式的运算中，如果 x 是一个事件，而且该引用发生在声明了 x 事件的类型之外，则这种运算结果的类型为 void（这正好与该运算的实际效果相反，它用于给 x 赋值，应该具有 x 所属的类型）。此规则能够禁止外部代码以间接方式来检查一个事件的基础委托。

以下示例显示如何将事件处理程序添加到 Button 类的实例：

```

public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
    public event EventHandler Click;
}
public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;
    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }
    void OkButtonClick(object sender, EventArgs e) {
        // 处理 OkButton.Click 事件
    }
    void CancelButtonClick(object sender, EventArgs e) {
        // 处理 CancelButton.Click 事件
    }
}

```

此处，LoginDialog 的实例构造函数创建两个 Button 实例并将事件处理程序附加到 Click 事件。

10.7.1 类似字段的事件

在包含事件声明的类或结构的程序文本内，某些事件可以像字段一样使用。若要以这样的方式使用，事件不能是 **abstract** 或 **extern**，而且不能显式地包含事件访问器声明。此类事件可以用在任何允许使用字段的上下文中。该字段含有一个委托 (§15)，它引用已添加到相应事件的事件处理程序列表。如果尚未添加任何事件处理程序，则该字段包含 **null**。

对于下面的示例：

```

public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
    public event EventHandler Click;
    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }
    public void Reset() {
        Click = null;
    }
}

```

Click 在 Button 类中用做一个字段。如上例所示，可以在委托调用表达式中检查、修改和使用字段。Button 类中的 OnClick 方法用于“引发”一个 Click 事件。“引发一个事件”与“调用一个由该事件表示的委托”完全等效，因此没有用于引发事件的特殊语言构造。请注意，在委托调用之前有一个检查，以确保该委托非空。

在 Button 类的声明外，Click 成员只能用在 += 和 -= 运算符的左边，如

```
b.Click += new EventHandler(...);
```

将一个委托追加到 Click 事件的调用列表，而

```
b.Click += new EventHandler(...);
```

则从 Click 事件的调用列表中移除一个委托。

当编译一个类似字段的事件时，编译器会自动创建一个存储区来存放相关的委托，并为事件创建相应的访问器以向委托字段中添加或移除事件处理程序。为了线程安全，添加或移除操作需在为实例事件的包含对象加锁 (§8.12) 的情况下进行，或者在为静态事件的类型对象 (§7.5.11) 加锁的情况下进行。

因此，下列形式的实例事件声明：

```
class X {
    public event D Ev;
}
```

可以编译为如下语句：

```
class X {
    private D __Ev; // 持有该委托的字段
    public event D Ev {
        add {
            lock(this) { __Ev = __Ev + value; }
        }
        remove {
            lock(this) { __Ev = __Ev - value; }
        }
    }
}
```

在类 X 中，引用 Ev 在编译时改为引用隐藏字段 _Ev。名称 “_Ev” 是任意的，隐藏字段可以具有任何名称或根本没有名称。

同样，如下形式的静态事件声明：

```
class X {
    public static event D Ev;
}
```

可以编译为如下语句：

```
class X {
    private static D __Ev; // 持有该委托的字段
    public static event D Ev {
        add {
            lock(typeof(X)) { __Ev = __Ev + value; }
        }
        remove {
            lock(typeof(X)) { __Ev = __Ev - value; }
        }
    }
}
```

10.7.2 事件访问器

事件声明通常省略事件访问器声明，如前面的 Button 示例中所示。但会有一些特殊

情况，例如，为每个事件设置一个字段所造成的内存开销，有时会变得不可接受。在这种情况下，可以在类中使用事件访问器声明，并采用专用机制来存储事件处理程序列表。

事件的事件访问器声明指定与添加和移除事件处理程序相关联的可执行语句。

访问器声明由一个添加访问器声明和一个移除访问器声明组成。每个访问器声明包含标记 `add` 或 `remove`，后跟一个块。与添加访问器声明相关联的块指定添加事件处理程序时要执行的语句，而与移除访问器声明相关联的块指定移除事件处理程序时要执行的语句。

每个添加访问器声明和移除访问器声明都相当于一个方法，它具有一个属于事件类型的值参数并且其返回类型为 `void`。事件访问器的隐式参数名为 `value`。当事件用在事件赋值中时，就会调用适当的事件访问器。具体说来，如果赋值运算符为 `+=`，则使用添加访问器，而如果赋值运算符为 `-=`，则使用移除访问器。在两种情况下，赋值运算符的右操作数都用做事件访问器的参数。添加访问器声明或移除访问器声明的块必须遵循§10.5.8 中所描述的适用于 `void` 方法的规则。具体说来，不允许此类块中的 `return` 语句指定表达式。

由于事件访问器隐式地具有一个名为 `value` 的参数，因此在事件访问器中声明的局部变量或常数若使用该名称，就会导致编译时错误。

对于下面的示例：

```
class Control: Component
{
    // 事件的惟一键
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();
    // 返回与键相关联的事件处理程序
    protected Delegate GetEventHandler(object key) {...}
    // 添加与键相关联的事件处理程序
    protected void AddEventHandler(object key, Delegate handler) {...}
    // 删除与键相关联的事件处理程序
    protected void RemoveEventHandler(object key, Delegate handler) {...}
    // MouseDown 事件
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }
    // MouseUp 事件
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }
    // 调用 MouseUp 事件
    protected void OnMouseUp(MouseEventArgs args) {
        MouseEventHandler handler;
        handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
        if (handler != null)
            handler(this, args);
    }
}
```

`Control` 类为事件实现了一个内部存储机制。`AddEventHandler` 方法将委托值与键关联，`GetEventHandler` 方法返回当前与键关联的委托，而 `RemoveEventHandler` 方法将移除一个委托，使它不再成为指定事件的一个事件处理程序。可以推断：在这样设计的基础存储机制下，当一个键所关联的委托值为 `null` 时，不会有存储开销，从而使未处理的事

件不占任何存储空间。

10.7.3 静态事件和实例事件

当事件声明包含 `static` 修饰符时，称该事件为**静态事件**（`static event`）；不存在 `static` 修饰符时，称该事件为**实例事件**（`instance event`）。

静态事件不和特定实例关联，因此在静态事件的访问器中引用 `this` 会导致编译时错误。

实例事件与类的给定实例关联，此实例在该事件的访问器中可以用 `this` (§7.5.7) 来访问。

在 `E.M` 形式的成员访问 (§7.5.4) 中引用事件时，如果 `M` 为静态事件，则 `E` 必须表示包含 `M` 的类型，如果 `M` 为实例事件，则 `E` 必须表示包含 `M` 的类型的一个实例。§10.2.5 对静态成员和实例成员之间的差异进行了进一步讨论。

10.7.4 虚拟、密封、重写和抽象访问器

`virtual` 事件声明指定事件的访问器是虚拟的。`virtual` 修饰符适用于事件的两个访问器。

`abstract` 事件声明指定事件的访问器是虚拟的，但是不提供这些访问器的实际实现。而且，非抽象派生类需要通过重写事件来提供它们自己的访问器实现。由于抽象事件声明的访问器不提供任何实际实现，因此它的访问器体只包含一个分号。

同时包含 `abstract` 和 `override` 修饰符的事件声明指定该事件是抽象的并重写一个基事件。此类事件的访问器也是抽象的。

只允许在抽象类 (§10.1.1.1) 中使用抽象事件声明。

一个被继承的虚拟事件的访问器可以在相关的派生类中用一个含有 `override` 修饰符的事件声明来进行重写，这称为重写事件声明。重写事件声明不声明新事件。实际上，它只是对现有虚拟事件的访问器的实现进行专用化。

重写事件声明必须采用与被重写事件完全相同的可访问性修饰符、类型和名称。

重写事件声明可以包含 `sealed` 修饰符。使用此修饰符可以防止相关的派生类进一步重写该事件。密封事件的访问器也是密封的。

重写事件声明包含 `new` 修饰符会导致编译时错误。

除了在声明和调用语法中的差异，虚拟、密封、重写和抽象访问器与虚拟、密封、重写和抽象方法具有完全相同的行为。准确地说，§10.5.3，§10.5.4，§10.5.5 和 §10.5.6 中描述的规则仍然适用，就好像访问器是相应形式的方法一样。每个访问器都对应于一个方法，它只有一个属于所涉及的事件类型的值参数、`void` 返回类型，以及与该事件相同的修饰符。

10.8 索引器

索引器（`indexer`）是这样一个成员：它使对象能够用与数组相同的方式进行索引。索

引器是使用索引器声明来声明的:

indexer-declaration: (索引器声明:)

attributes_{opt} indexer-modifiers_{opt} indexer-declarator { accessor-declarations } (属性_{可选}
索引器修饰符_{可选} 索引器声明符 { 访问器声明 })

indexer-modifiers: (索引器修饰符:)

indexer-modifier (索引器修饰符)

indexer-modifiers indexer-modifier (索引器修饰符 索引器修饰符)

indexer-modifier: (索引器修饰符:)

new
public
protected
internal
private
virtual
sealed
override
abstract
extern

indexer-declarator: (索引器声明符:)

type this [formal-parameter-list] (类型 this [形参表])

type interface-type . this [formal-parameter-list] (类型 接口类
型 . this [形参表])

索引器声明可包含一组特性 (§17) 和 4 个访问修饰符 (§10.2.3) 的有效组合, 以及 **new** (§10.2.2), **virtual** (§10.5.3), **override** (§10.5.4), **sealed** (§10.5.5), **abstract** (§10.5.6) 和 **extern** (§10.5.7) 修饰符。

关于有效的修饰符组合, 事件声明与方法声明 (§10.5) 遵循相同的规则 (惟一的例外是: 在索引器声明中不允许使用静态修饰符)。

修饰符 **virtual**, **override** 和 **abstract** 是互相排斥的, 但有一种情况除外: **abstract** 和 **override** 修饰符可以一起使用以便抽象索引器可以重写虚拟索引器。

索引器声明的类型用于指定由该声明引入的索引器的元素类型。除非索引器是一个显式接口成员的实现, 否则该类型后要跟一个关键字 **this**。而对于显式接口成员的实现, 该类型后要先跟一个接口类型、一个 “.”, 再跟一个关键字 **this**。与其他成员不同的是, 索引器不具有用户定义的名称。

形参表用于指定索引器的参数。索引器的形参表对应于方法的形参表 (§10.5.1), 不同之处仅在于索引器的形参表中必须至少含有一个参数, 并且不允许使用 **ref** 和 **out** 参数修饰符。

索引器的类型和在形参表中引用的每个参数的类型都必须至少具有与索引器本身相同的可访问性 (§3.5.4)。

访问器声明 (§10.6.2) (它必须被括在 “{}” 标记内) 用于声明该索引器的访问器。这些访问器用来指定与读取和写入索引器元素相关联的可执行语句。

虽然访问索引器元素的语法与访问数组元素的语法相同，但是索引器元素并不属于变量。因此，不可能将索引器元素作为 `ref` 或 `out` 参数传递。

索引器的形参表定义索引器的签名 (§3.6)。具体说来，索引器的签名由其形参的数量和类型组成，但索引器元素的类型和形参的名称都不是索引器签名的组成部分。

索引器的签名必须不同于在同一个类中声明的所有其他索引器的签名。

索引器和属性在概念上非常类似，但在下列方面有所区别：

- 属性由它的名称标识，而索引器由它的签名标识。
- 属性是通过简单名称 (§7.5.2) 或成员访问 (§7.5.4) 来访问的，而索引器元素则是通过元素访问 (§7.5.6.2) 来访问的。
- 属性可以是 `static` 成员，而索引器始终是实例成员。
- 属性的 `get` 访问器对应于不带参数的方法，而索引器的 `get` 访问器对应于与索引器具有相同的形参表的方法。
- 属性的 `set` 访问器对应于具有名为 `value` 的单个参数的方法，而索引器的 `set` 访问器对应于与索引器具有相同的形参表加上一个名为 `value` 的附加参数的方法。
- 若在索引器访问器内使用与该索引器的参数相同的名称来声明局部变量，就会导致编译时错误。
- 在重写属性声明中，被继承的属性是使用语法 `base.P` 访问的，其中 `P` 为属性名称。在重写索引器声明中，被继承的索引器是使用语法 `base[E]` 访问的，其中 `E` 是一个用逗号分隔的表达式列表。

除上述差异以外，所有在 §10.6.2 和 §10.6.3 中定义的规则都适用于索引器访问器及属性访问器。

当索引器声明包含 `extern` 修饰符时，称该索引器为外部索引器 (`external indexer`)。因为外部索引器声明不提供任何实际的实现，所以它的每个访问器声明都由一个分号组成。

下面的示例声明了一个 `BitArray` 类，该类实现了一个索引器，用于访问位数组中的单个位。

```
using System;
class BitArray
{
    int[] bits;
    int length;
    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }
    public int Length {
        get { return length; }
    }
    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
    }
}
```

```

    set {
        if (index < 0 || index >= length) {
            throw new IndexOutOfRangeException();
        }
        if (value) {
            bits[index >> 5] |= 1 << index;
        }
        else {
            bits[index >> 5] &= ~(1 << index);
        }
    }
}
}

```

BitArray 类的一个实例所占的内存远小于相应的 **bool[]** (这是由于前者的每个值只占一位, 而后的每个值要占一个字节), 而且, 它可以执行与 **bool[]** 相同的操作。

下面的 **CountPrimes** 类使用 **BitArray** 和经典的“筛选”算法计算 1 和给定的最大数之间质数的数目:

```

class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }
    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}

```

请注意, 访问 **BitArray** 的元素的语法与访问 **bool[]** 的元素的语法完全相同。

下面的示例显示一个具有带两个参数的索引器的 **26×10** 网格类。第一个参数必须是 **A~Z** 范围内的大写或小写字母, 而第二个参数必须是 **0~9** 范围内的整数。

```

using System;
class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;
    int[,] cells = new int[NumRows, NumCols];
    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
        }
    }
}

```

```

        return cells[c - 'A', col];
    }
    set {
        c = Char.ToUpper(c);
        if (c < 'A' || c > 'Z') {
            throw new ArgumentException();
        }
        if (col < 0 || col >= NumCols) {
            throw new IndexOutOfRangeException();
        }
        cells[c - 'A', col] = value;
    }
}

```

§7.4.2 中描述了索引器重载决策规则。

10.9 运算符

运算符 (operator) 是一种用来定义可应用于类的实例的表达式运算符含义的成员。运算符是使用运算符声明来声明的:

operator-declaration: (运算符声明:)

attributes_{opt} operator-modifiers operator-declarator operator-body (属性_{可选} 运算符
修饰符 运算符声明符 运算符体)

operator-modifiers: (运算符修饰符:)

operator-modifier (运算符修饰符)

operator-modifiers operator-modifier (运算符修饰符 运算符修饰符)

operator-modifier: (运算符修饰符:)

```

public
static
extern

```

operator-declarator: (运算符声明符:)

unary-operator-declarator (一元运算符声明符)

binary-operator-declarator (二元运算符声明符)

conversion-operator-declarator (转换运算符声明符)

unary-operator-declarator: (一元运算符声明符:)

type operator overloadable-unary-operator (type identifier) (类 型

operator 可重载的一元运算符 (类型 标识符))

overloadable-unary-operator: one of (可重载的一元运算符: 下列之一)

+ - ! ~ ++ -- true false

binary-operator-declarator: (二元运算符声明符:)

type operator overloadable-binary-operator (type identifier , type identifier)

(类型 operator 可重载的二元运算符 (类型 标识符 , 类型 标识符))

overloadable-binary-operator: one of (可重载的二元运算符: 下列之一)

+ - * / % & | ^ << >> == != > < >= <=

conversion-operator-declarator: (转换运算符声明符:)

implicit operator type (type identifier) (implicit operator 类型 (类型 标识符))

explicit operator type (type identifier) (explicit operator 类型 (类型 标识符))

operator-body: (运算符体:)

block (块)

;

有三个类别的可重载运算符：一元运算符 (§10.9.1)、二元运算符 (§10.9.2) 和转换运算符 (§10.9.3)。

当运算符声明包含 `extern` 修饰符时，称该运算符为外部运算符 (`external operator`)。因为外部运算符不提供任何实际的实现，所以它的运算符体由一个分号组成。对于所有其他运算符，运算符体由一个块组成，它指定在调用该运算符时需要执行的语句。运算符的块必须遵循 §10.5.8 中所描述的适用于值返回方法的规则。

下列规则适用于所有的运算符声明。

- 运算符声明必须同时包含一个 `public` 和一个 `static` 修饰符。
- 运算符的参数必须是值参数。在运算符声明中指定 `ref` 或 `out` 参数会导致编译时错误。
- 运算符的签名 (§10.9.1、§10.9.2、§10.9.3) 必须不同于在同一类中声明的所有其他运算符的签名。
- 运算符声明中引用的所有类型必须至少具有与运算符本身一样的可访问性 (§3.5.4)。
- 同一修饰符在一个运算符声明中多次出现是错误的。

每个运算符类别都有附加的限制，将在下面几节中说明。

与其他成员一样，在基类中声明的运算符由派生类继承。由于运算符声明始终要求声明运算符的类或结构参与运算符的签名，因此在派生类中声明的运算符不可能隐藏在基类中声明的运算符。所以，运算符声明中永远不会要求也不允许使用 `new` 修饰符。

可以在 §7.2 中找到关于一元和二元运算符的其他信息。

可以在 §6.4 中找到关于转换运算符的其他信息。

10.9.1 一元运算符

下列规则适用于一元运算符声明，其中 `T` 表示包含运算符声明的类或结构类型：

- 一元 `+`、`-`、`!` 或 `~` 运算符必须带有单个 `T` 类型的参数，并且可以返回任何类型。
- 一元 `++` 或 `--` 运算符必须带有单个 `T` 类型的参数且必须返回类型 `T`。
- 一元 `true` 或 `false` 运算符必须带有单个 `T` 类型的参数并且必须返回类型 `bool`。

一元运算符的签名包含运算符标记 (`+`、`-`、`!`、`~`、`++`、`--`、`true` 或 `false`) 和单个形

参的类型。返回类型和形参的名称不是一元运算符的签名的组成部分。

一元运算符 `true` 和 `false` 要求成对的声明。如果类只声明了这两个运算符的其中一个而没有声明另一个，将发生编译时错误。§7.11.2 和 §7.16 中对 `true` 运算符和 `false` 运算符进行了进一步描述。

下面的示例显示了对一个整数向量类的 `operator ++` 的实现，以及随后对它的使用：

```
public class IntVector
{
    public IntVector(int length) {...}
    public int Length {...}           // 只读属性
    public int this[int index] {...}   // 只写索引器
    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}
class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4); // 4 x 0 向量
        IntVector iv2;
        iv2 = iv1++; // iv2 包含 4 x 0, iv1 包含 4 x 1
        iv2 = ++iv1; // iv2 包含 4 x 2, iv1 包含 4 x 2
    }
}
```

请注意，此运算符方法怎样返回通过向操作数添加 1 而产生的值，就像后缀增量和减量运算符 (§7.5.9)，以及前缀增量和减量运算符 (§7.6.5) 一样。与在 C++ 中不同，此方法并不需要直接修改其操作数的值。实际上，修改操作数的值会违反后缀增量运算符的标准语义。

10.9.2 二元运算符

二元运算符必须带两个参数，其中至少有一个必须具有声明了该运算符的类或结构类型。移位运算符 (§7.8) 的参数被进一步限制。二元运算符可以返回任何类型。

二元运算符的签名由运算符标记 (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=` 或 `<=`) 和两个形参的类型组成。它本身的返回类型及形参的名称不是二元运算符签名的组成部分。

某些二元运算符要求成对声明。对于需要成对声明的运算符，若声明了其中一个，就必须对另一个做出相匹配的声明。当两个运算符声明具有相同的返回类型且各个参数具有相同的类型时，则它们相匹配。下列运算符要求成对声明：

- 运算符 `==` 和运算符 `!=`
- 运算符 `>` 和运算符 `<`
- 运算符 `>=` 和运算符 `<=`

10.9.3 转换运算符

转换运算符声明引入用户定义的转换 (§6.4)，此转换可以扩充预定义的隐式转换和显式转换。

包含 `implicit` 关键字的转换运算符声明引入用户定义的隐式转换。隐式转换可以在多种情况下发生，包括函数成员调用、强制转换表达式和赋值。§6.1 中对此进行了进一步描述。

包含 `explicit` 关键字的转换运算符声明引入用户定义的显式转换。显式转换可以发生在强制转换表达式中，§6.2 中对此进行了进一步描述。

转换运算符从某个源类型（即该转换运算符的参数类型）转换到一个目标类型（即该转换运算符的返回类型）。如果下列条件都为真，则允许类或结构声明一个从源类型 `S` 到目标类型 `T` 的转换：

- `S` 和 `T` 是不同的类型。
- `S` 和 `T` 中总有一个是声明了该运算符的类类型或结构类型。
- `S` 和 `T` 都不是 `object` 或接口类型。
- `T` 不是 `S` 的基类，`S` 也不是 `T` 的基类。

从第二条规则可以推知，转换运算符必须将声明了该运算符的类或结构类型或者作为目标类型，或者作为源类型。例如，一个类或结构类型 `C` 可以定义从 `C` 到 `int` 和从 `int` 到 `C` 的转换，但是不能定义从 `int` 到 `bool` 的转换。

不能重新定义一个已存在的预定义转换。因此，不允许转换运算符将 `object` 转换为其他类型或将其他类型转换为 `object`，这是因为已存在一些隐式和显式转换来执行 `object` 和所有其他类型之间的转换。同样，转换的源类型和目标类型不能是对方的基类型，这是因为已经存在了这样的转换。

用户定义的转换不能用于在接口类型和其他类型之间进行转换。具体说来，这条规定确保了在转换为接口类型时不会发生任何用户定义的转换，以及只有在被转换的对象实际上实现了指定的接口类型时，到该接口类型的转换才会成功。

转换运算符的签名由源类型和目标类型组成（请注意，这是惟一一种其返回类型参与签名的成员形式）。转换运算符的 `implicit` 类别或 `explicit` 类别不是运算符签名的组成部分。因此，类或结构不能同时声明具有相同的源类型和目标类型的 `implicit` 和 `explicit` 转换运算符。

一般来说，如果设计一个用户定义的隐式转换，就应当确保执行该转换时绝不会引发异常，和丢失信息。如果用户定义的转换可能导致引发异常（例如，由于源参数超出范围）或丢失信息（如放弃高序位），则该转换应该定义为显式转换。

对于下面的示例：

```
using System;
public struct Digit
{
    byte value;
    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
    }
}
```

```

        this.value = value;
    }
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}

```

从 Digit 到 byte 的转换是隐式的，这是因为它永远不会抛出异常或丢失信息；但是从 byte 到 Digit 的转换是显式的，这是由于 Digit 只能表示 byte 的可能值的一个子集。

10.10 实例构造函数

实例构造函数（instance constructor）是实现初始化类实例所需操作的成员。实例构造函数是使用构造函数声明来声明的：

constructor-declaration:（构造函数声明：）

attributes_{opt} constructor-modifiers_{opt} constructor-declarator constructor-body（属性_{可选} 构造函数修饰符_{可选} 构造函数声明符 构造函数体）

constructor-modifiers:（构造函数修饰符：）

constructor-modifier（构造函数修饰符）

constructor-modifiers constructor-modifier（构造函数修饰符 构造函数修饰符）

constructor-modifier:（构造函数修饰符：）

```

public
protected
internal
private
extern

```

constructor-declarator:（构造函数声明符：）

identifier（ formal-parameter-list_{opt} ） constructor-initializer_{opt}（标识符（形参表_{可选}）构造函数初始值设定项_{可选}）

constructor-initializer:（构造函数初始值设定项：）

: base（ argument-list_{opt} ）(: base（参数列表_{可选}））

: this（ argument-list_{opt} ）(: this（参数列表_{可选}））

constructor-body:（构造函数体：）

block（块）

;

构造函数声明可包含一组特性（§17）、4 个访问修饰符（§10.2.3）的有效组合和一个 extern（§10.5.7）修饰符。在一个构造函数声明中同一修饰符不能多次出现。

构造函数声明符中的标识符必须是声明了该实例构造函数的那个类的名称。如果指定了任何其他名称，则发生编译时错误。

可选的实例构造函数的形参表必须遵循与方法 (§10.5) 形参表同样的规则。此形参表定义实例构造函数的签名 (§3.6)，并且在函数调用中控制重载决策 (§7.4.2) 过程以选择某个特定实例的构造函数。

在实例构造函数的形参表中引用的各个类型必须至少具有与构造函数本身相同的可访问性 (§3.5.4)。

可选的构造函数初始值设定项用于指定在执行此实例构造函数的构造函数体中给出的语句之前需要调用的另一个实例构造函数。§10.10.1 对此有进一步的描述。

当构造函数声明中包含 `extern` 修饰符时，称该构造函数为外部构造函数 (`external constructor`)。因为外部构造函数声明不提供任何实际的实现，所以它的构造函数体仅由一个分号组成。对于所有其他构造函数，构造函数体都由一个块组成，它用于指定初始化该类的一个新实例时需要执行的语句。这正好相当于一个具有 `void` 返回类型的实例方法的块 (§10.5.8)。

实例构造函数是不能被继承的。因此，类除了自己声明的实例构造函数外，不可能有其他的实例构造函数。如果一个类没有声明任何实例构造函数，则会自动地为它提供一个默认的实例构造函数 (§10.10.4)。

实例构造函数是由对象创建表达式 (§7.5.10.1) 通过构造函数初始值设定项调用的。

10.10.1 构造函数初始值设定项

除了类 `object` 的实例构造函数以外，所有其他的实例构造函数都隐式地包含一个对另一个实例构造函数的调用，该调用紧靠在构造函数体的前面。要隐式调用的构造函数是由构造函数初始值设定项确定的：

- **base** (参数列表_{可选}) 形式的实例构造函数初始值设定项导致调用直接基类中的实例构造函数。该构造函数是根据参数列表和 §7.4.2 的重载决策规则选择的。候选实例构造函数集由直接基类中包含的所有可访问实例构造函数组成 (包括所有默认构造函数，详见 §10.10.4 中的定义)。如果此集合为空，或者无法标识单个最佳实例构造函数，就会发生编译时错误。
- **this** (参数列表_{可选}) 形式的实例构造函数初始值设定项导致调用该类本身所声明的实例构造函数。构造函数是根据参数列表和 §7.4.2 的重载决策规则选择的。候选实例构造函数集包括在该类中声明的所有可访问的实例构造函数。如果此集合为空，或者无法标识单个最佳实例构造函数，就会发生编译时错误。如果实例构造函数声明中包含调用构造函数本身的构造函数初始值设定项，也会发生编译时错误。

如果一个实例构造函数中没有构造函数初始值设定项，则隐式地添加一个 `base()` 形式的构造函数初始值设定项。因此，下列形式的实例构造函数声明

```
C(...) {...}
```

完全等效于

```
C(...): base() {...}
```

实例构造函数声明中的形参表所给出的参数范围包含该声明的实例构造函数初始值设定项。因此，构造函数初始值设定项可以访问该构造函数的参数。例如：

```
class A
{
    public A(int x, int y) {}
}
class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

实例构造函数初始值设定项不能访问正在创建的实例。因此在构造函数初始值设定项的参数表达式中引用 `this` 会导致编译时错误，就像参数表达式通过简单名称引用任何实例成员会导致编译时错误一样。

10.10.2 实例变量初始值设定项

如果实例构造函数没有构造函数初始值设定项，或仅具有 `base(...)` 形式的构造函数初始值设定项，该构造函数就会隐式地执行在该类中声明的实例字段的初始化操作，这些操作由对应的字段声明中的变量初始值设定项指定。这对应于一个赋值序列，它们会在进入构造函数时，在对直接基类的构造函数进行隐式调用之前立即执行。这些变量初始值设定项按它们出现在类声明中的文本顺序执行。

10.10.3 构造函数执行

变量初始值设定项被转换为赋值语句，而这些语句将在对基类实例构造函数进行调用之前执行。这种排序确保了在执行任何访问该实例的语句之前，所有实例字段都已按照它们的变量初始值设定项进行了初始化。

对于给定示例：

```
using System;
class A
{
    public A() {
        PrintFields();
    }
    public virtual void PrintFields() {}
}
class B: A
{
    int x = 1;
    int y;
    public B() {
        y = -1;
    }
    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```

当使用 `new B()` 创建 `B` 的实例时，会产生如下输出：

```
x = 1, y = 0
```

`x` 的值为 1，这是由于变量初始值设定项是在调用基类实例构造函数之前执行的。但是，`y` 的值为 0（`int` 型变量的默认值），这是因为对 `y` 的赋值直到基类构造函数返回之后才执行。

可以这样设想来帮助理解：将实例变量初始值设定项和构造函数初始值设定项视为自动插入到构造函数体之前的语句。示例：

```
using System;
using System.Collections;
class A
{
    int x = 1, y = -1, count;
    public A() {
        count = 0;
    }
    public A(int n) {
        count = n;
    }
}
class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;
    public B(): this(100) {
        items.Add("default");
    }
    public B(int n): base(n - 1) {
        max = n;
    }
}
```

包含若干个变量初始值设定项，还包含两种形式（`base` 和 `this`）的构造函数初始值设定项。此示例对应于下面显示的代码，其中每个注释指示一个自动插入的语句（用于自动插入的构造函数调用的语法是无效的，只是用来阐释此机制）。

```
using System.Collections;
class A
{
    int x, y, count;
    public A() {
        x = 1;                // 变量初始化
        y = -1;               // 变量初始化
        object();              // 调用 object() 构造函数
        count = 0;
    }
    public A(int n) {
        x = 1;                // 变量初始化
        y = -1;               // 变量初始化
        object();              // 调用 object() 构造函数
        count = n;
    }
}
class B: A
```

```
{
    double sqrt2;
    ArrayList items;
    int max;
    public B(): this(100) {
        B(100); //调用 B(int) 构造函数
        items.Add("default");
    }
    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0); //变量初始化
        items = new ArrayList(100); //变量初始化
        A(n - 1); //调用 A(int) 构造函数
        max = n;
    }
}
```

10.10.4 默认构造函数

如果一个类不包含任何实例构造函数声明，则会自动地为该类提供一个默认实例构造函数。默认构造函数只是调用直接基类的无参数构造函数。如果直接基类没有可访问的无参数实例构造函数，则发生编译时错误。对于抽象类，它的默认构造函数的声明可访问性是受保护的。而对于非抽象类，它的默认构造函数的声明可访问性是公共的。因此，默认构造函数始终为下列形式：

```
protected C(): base() {}
```

或者

```
public C(): base() {}
```

其中 C 为类的名称。

对于下面的示例：

```
class Message
{
    object sender;
    string text;
}
```

由于类不包含任何实例构造函数声明，因此为它提供了一个默认构造函数。因而，此示例完全等效于

```
class Message
{
    object sender;
    string text;
    public Message(): base() {}
}
```

10.10.5 私有构造函数

当类 T 只声明了私有实例构造函数时，在 T 的程序文本外部，既不可能从 T 派

生出新的类，也不可能直接创建 `T` 的任何实例。因此，如果想要设计一个类，它只包含静态成员而且不想使它被实例化，则只需给它添加一个空的私有实例构造函数，即可达到目的。例如：

```
public class Trig
{
    private Trig() {} // 阻止实例化
    public const double PI = 3.14159265358979323846;
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

`Trig` 类用于将相关的方法和常数组合在一起，但是它不能被实例化。因此它声明了单个空的私有实例构造函数。若要取消默认构造函数的自动生成，则必须至少声明一个实例构造函数。

10.10.6 可选的实例构造函数参数

`this(...)` 形式的构造函数初始值设定项通常与重载一起使用，以实现可选的实例构造函数。对于下面的示例：

```
class Text
{
    public Text(): this(0, 0, null) {}
    public Text(int x, int y): this(x, y, null) {}
    public Text(int x, int y, string s) {
        // 实际构造函数的实现
    }
}
```

前两个实例构造函数只为调用中没有传递过来的参数提供相应的默认值。这两个构造函数都使用 `this(...)` 构造函数初始值设定项来调用实际完成初始化新实例工作的第三个实例构造函数。这样，实际效果就是该实例构造函数具有可选的参数：

```
Text t1 = new Text(); // 相当于 Text(0, 0, null)
Text t2 = new Text(5, 10); // 相当于 (5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

10.11 静态构造函数

静态构造函数（static constructor）是一种用于实现初始化类所需操作的成员。静态构造函数是使用静态构造函数声明来声明的：

static-constructor-declaration:（静态构造函数声明：）

attributes_{opt} static-constructor-modifiers identifier () static-constructor-body (属性_{可选} 静态构造函数修饰符 标识符 () 静态构造函数体)

static-constructor-modifiers:（静态构造函数修饰符：）

```
extern_opt static
static extern_opt
```

static-constructor-body: (静态构造函数体:)

block (块)

;

静态构造函数声明可包含一组特性 (§17) 和一个 `extern` (§10.5.7) 修饰符。

静态构造函数声明的标识符必须是声明了该静态函数的那个类的名称。如果指定了任何其他名称, 则发生编译时错误。

当静态构造函数声明包含 `extern` 修饰符时, 称该静态构造函数为外部静态构造函数。因为外部静态构造函数声明不提供任何实际的实现, 所以它的静态构造函数体由一个分号组成。对于所有其他的静态构造函数声明, 静态构造函数体都是一个块, 它指定当初始化该类时需要执行的语句。这正好相当于具有 `void` 返回类型的静态方法的方法体 (§10.5.8)。

静态构造函数是不可继承的, 而且不能被直接调用。

类的静态构造函数在给定应用程序域中至多执行一次。应用程序域中第一次发生以下事件时将触发静态构造函数的执行:

- 创建类的实例。
- 引用类的任何静态成员。

如果类中包含用来开始执行的 `Main` 方法 (§3.1), 则该类的静态构造函数将在调用 `Main` 方法之前执行。如果类包含任何带有初始值设定项的静态字段, 则在执行该类的静态构造函数时, 先要按照文本顺序执行那些初始值设定项。

示例:

```
using System;
class Test
{
    static void Main() {
        A.F();
        B.F();
    }
}
class A
{
    static A() {
        Console.WriteLine("Init A");
    }
    public static void F() {
        Console.WriteLine("A.F");
    }
}
class B
{
    static B() {
        Console.WriteLine("Init B");
    }
    public static void F() {
        Console.WriteLine("B.F");
    }
}
```

一定产生输出:

```
Init A
A.F
Init B
B.F
```

这是因为 A 的静态构造函数的执行是通过调用 A.F 触发的, 而 B 的静态构造函数的执行是通过调用 B.F 触发的。

上述过程有可能构造出循环依赖关系, 其中, 带有变量初始值设定项的静态字段能够在其处于默认值状态时被观测到。

示例:

```
using System;
class A
{
    public static int X;
    static A() {
        X = B.Y + 1;
    }
}
class B
{
    public static int Y = A.X + 1;
    static B() {}
    static void Main() {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}
```

产生下列输出:

```
X = 1, Y = 2
```

为执行 Main 方法, 系统在运行类 B 的静态构造函数之前首先要运行 B.Y 的初始值设定项。因为引用了 A.X 的值, 所以 Y 的初始值设定项导致运行 A 的静态构造函数。这样, A 的静态构造函数将继续计算 X 的值, 从而获取 Y 的默认值 0, 而 A.X 被初始化为 1。这样就完成了运行 A 的静态字段初始值设定项和静态构造函数的进程, 控制返回到 Y 的初始值的计算, 计算结果变为 2。

10.12 析构函数

析构函数 (destructor) 是一种用于实现析构类实例所需操作的成员。析构函数是用析构函数的声明来声明的:

destructor-declaration: (析构函数声明:)

attributes_{opt} extern_{opt} ~ identifier () destructor-body (属性_{可选} extern_{可选} ~ 标识符 () 析构函数体)

destructor-body: (析构函数体:)

block (块)

;

析构函数声明可以包括一组特性 (§17)。

析构函数声明的标识符必须是声明了该析构函数的那个类的名称。如果指定了任何其他名称，则发生编译时错误。

当析构函数声明包含 `extern` 修饰符时，称该析构函数为外部析构函数 (`external destructor`)。因为外部析构函数声明不提供任何实际的实现，所以它的析构函数体由一个分号组成。对于所有其他析构函数，析构函数体都包含一个块，它指定当销毁该类的一个实例时需要执行的语句。析构函数体完全相当于具有 `void` 返回类型的实例方法的方法体 (§10.5.8)。

析构函数是不可被继承的。因此，除了自己所声明的析构函数外，一个类不具有其他析构函数。

由于析构函数要求不能带有参数，因此它不能被重载，所以一个类至多只能有一个析构函数。

析构函数是被自动调用的，它不能被显式调用。当任何代码都不再可能使用一个实例时，该实例就符合被销毁的条件。此后，它所对应的实例析构函数随时都可能被调用。销毁实例时，按照从派生程度最大到派生程度最小的顺序，调用该实例的继承链中的各个析构函数。析构函数可以在任何线程上执行。有关控制何时及如何执行析构函数的规则的进一步讨论，请参见§3.9。

下列示例：

```
using System;
class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}
class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}
class Test
{
    static void Main() {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

的输出是这样的：

```
B's destructor
A's destructor
```

这是由于继承链中的析构函数是按照从派生程度最大到派生程度最小的顺序调用的。

析构函数实际上是重写了 `System.Object` 中的虚方法 `Finalize`。C# 程序中不允许重写

此方法或直接调用它（或它的重写）。例如，下列程序：

```
class A
{
    override protected void Finalize() {} // 错误
    public void F() {
        this.Finalize(); // 错误
    }
}
```

包含两个错误。

编译器表现为好像此方法和它的重写根本不存在一样。因此，以下程序：

```
class A
{
    void Finalize() {} // 允许
}
```

是有效的，所声明的方法隐藏了 `System.Object` 的 `Finalize` 方法。

有关析构造函数引发异常时的表现的讨论，请参见§16.3。

第 11 章 结构

结构与类很相似，都表示可以包含数据成员和函数成员的数据结构。但是，与类不同的是，结构是一种值类型，并且不需要堆分配。结构类型的变量直接包含结构的数据，而类类型的变量包含对数据的引用（后者称为对象）。

结构对于具有值语义的小的数据结构特别有用。复数、坐标系中的点或字典中的“键-值”对都是结构的典型示例。这些数据结构的关键之处在于：它们只有少量数据成员，不要求使用继承或引用标识，而且它们适合使用值语义（赋值时直接复制值而不是复制它的引用）方便地实现。

正如§4.1.4中所描述的，C# 提供的简单类型，如 `int`、`double` 和 `bool`，实际上全都是结构类型。正如这些预定义类型是结构一样，也可以使用结构和运算符重载在 C# 语言中实现新的“基元”类型。在本章结尾 (§11.4) 给出了这种类型的两个示例。

11.1 结构声明

结构声明是一种用于声明新结构的类型声明 (§9.5)：

`struct-declaration`: (结构声明:)

`attributesopt struct-modifiersopt struct identifier struct-interfacesopt struct-body ;opt` (属性_{可选} 结构修饰符_{可选} `struct` 标识符 结构接口_{可选} 结构体 ;_{可选})

结构声明包含一组可选的特性 (§17)，后跟一组可选的结构修饰符 (§11.1.1)，再跟关键字 `struct` 和一个命名结构的标识符，然后跟一个可选的结构接口规范 (§11.1.2)，最后跟一个结构体 (§11.1.3)，根据需要后面还可以跟一个分号。

11.1.1 结构修饰符

结构声明可以根据需要包含一个结构修饰符序列：

`struct-modifiers`: (结构修饰符:)

`struct-modifier` (结构修饰符)

`struct-modifiers struct-modifier` (结构修饰符 结构修饰符)

`struct-modifier`: (结构修饰符:)

`new`
`public`
`protected`
`internal`
`private`

同一修饰符在结构声明中出现多次会导致编译时错误。

结构声明的修饰符与类声明 (§10.1.1) 的修饰符具有相同的意义。

11.1.2 结构接口

结构声明中可以含有一个结构接口规范，这种情况下称该结构实现给定的接口类型。

struct-interfaces: (结构接口:)

: **interface-type-list** (: 接口类型列表)

§13.4 对接口实现进行了进一步讨论。

11.1.3 结构体

结构的结构体用于定义该结构所包含的成员:

struct-body: (结构体:)

{ **struct-member-declarations_{opt}** } ({ 结构成员声明_{可选} })

11.2 结构成员

结构的成员由两部分组成: 由结构成员声明引入的成员, 以及从类型 `System.ValueType` 继承的成员。

struct-member-declarations: (结构成员声明:)

struct-member-declaration (结构成员声明)

struct-member-declarations struct-member-declaration (结构成员声明 结构成员声明)

struct-member-declaration: (结构成员声明:)

constant-declaration (常数声明)

field-declaration (字段声明)

method-declaration (方法声明)

property-declaration (属性声明)

event-declaration (事件声明)

indexer-declaration (索引器声明)

operator-declaration (运算符声明)

constructor-declaration (构造函数声明)

static-constructor-declaration (静态构造函数声明)

type-declaration (类型声明)

除了在 §11.3 中指出的区别外, 在从 §10.2 到 §10.11 中关于类成员的说明也适用于结构成员。

11.3 类和结构的区别

结构在以下几个方面和类是不同的：

- 结构是值类型 (§11.3.1)。
- 所有结构类型隐式地从类 `System.ValueType` 继承 (§11.3.2)。
- 对结构类型变量进行赋值意味着将创建所赋的值的一个“副本” (§11.3.3)。
- 结构的默认值的计算如下：将所有值类型字段设置为它们的默认值，并将所有引用类型字段设置为 `null`，这样就产生了该结构的默认值 (§11.3.4)。
- 使用装箱和取消装箱操作在结构类型和 `object` 之间进行转换 (§11.3.5)。
- 对于结构，`this` 的意义不同 (§11.3.6)。
- 在结构中，实例字段声明中不能含有变量初始值设定项 (§11.3.7)。
- 在结构中不能声明无参数的实例构造函数 (§11.3.8)。
- 在结构中不能声明析构函数 (§11.3.9)。

11.3.1 值语义

结构是值类型 (§4.1)，并且被称为具有值语义。另一方面，类是引用类型 (§4.2) 且被称为具有引用语义。

结构类型的变量直接包含了该结构的数据，而类类型的变量所包含的只是对相应数据的一个引用（被引用的数据称为“对象”）。

对于类，两个变量可能引用同一对象，因此对一个变量进行的操作可能影响另一个变量所引用的对象。对于结构，每个变量都有它们自己的数据副本（除 `ref` 和 `out` 参数变量外），因此对一个变量的操作不可能影响其他变量。另外，由于结构不是引用类型，因此结构类型的值不可能为 `null`。

给定下列声明：

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

代码片段：

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

输出值 10。将 `a` 赋给 `b` 将创建该值的一个副本，因此 `b` 不会受到对 `a.x` 进行的赋值的影响。假如 `Point` 被改而声明为类，则由于 `a` 和 `b` 将引用同一对象，因此输出将为 100。

11.3.2 继承

所有结构类型都隐式地从类 `System.ValueType` 继承，而后者则从类 `object` 继承。结构声明可以指定实现的接口列表，但是不能指定基类。

结构类型永远不会是抽象的，并且始终是隐式密封的。因此在结构声明中不允许使用 `abstract` 和 `sealed` 修饰符。

由于对结构而言不支持继承，所以结构成员的声明可访问性不能是 `protected` 或 `protected internal`。

结构中的函数成员不能是 `abstract` 或 `virtual`，因而 `override` 修饰符只适用于重写从 `System.ValueType` 继承的方法。

11.3.3 赋值

对结构类型变量的赋值创建了被赋值的一个“副本”。这不同于对类类型变量的赋值，后者所复制的是引用，而不是由该引用所标识的对象。

与赋值类似，将结构作为值参数传递或者作为函数成员的结果返回时，也创建了该结构的一个副本。但是，一个结构仍可作为 `ref` 或 `out` 参数（在函数成员调用中）被引用传递。

当结构的属性或索引器是赋值的目标时，与属性或索引器访问关联的实例表达式必须为变量类别。如果该实例表达式属于值类型，则发生编译时错误。§7.13.1 对此进行了详细的描述。

11.3.4 默认值

如§5.2 中所描述，有几种变量在创建时自动初始化为它们的默认值。对于类类型和其他引用类型的变量，此默认值为 `null`。但是，由于结构是不能为 `null` 的值类型，因此结构的默认值是通过将所有值类型字段设置为它们的默认值，并将所有引用类型字段设置为 `null` 而产生的值。

引用上面声明的 `Point` 结构，下面的示例

```
Point[] a = new Point[100];
```

将数组中的每个 `Point` 初始化为通过将 `x` 和 `y` 字段设置为零而产生的值。

结构的默认值相当于该结构的默认构造函数所返回的值 (§4.1.1)。与类不同，结构不允许声明无参数实例构造函数。相反，每个结构隐式地具有一个无参数实例构造函数，该构造函数始终返回相同的值，即通过将所有的值类型字段设置为它们的默认值，并将所有引用类型字段设置为 `null` 而得到的值。

设计一个结构时，要设法确保它的默认初始化状态是有效的状态。对于下面的示例：

```
using System;
struct KeyValuePair
```

```
{
    string key;
    string value;
    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

用户定义的实例构造函数不允许出现 `null` 值（除非在显式调用时）。但 `KeyValuePair` 变量可能会被初始化为它的默认值，这样，`key` 和 `value` 字段就都为 `null`，所以，设计该结构时，必须正确处理好此问题。

11.3.5 装箱和取消装箱

类类型的值可以被转换为 `object` 类型或由该类实现的接口类型，这只需在编译时把对应的引用当做另一个类型处理即可。与此类似，`object` 类型的值或者接口类型的值也可以被转换回类类型而不必更改相应的引用。当然，在这种情况下，需要进行“运行时类型检查”。

由于结构不是引用类型，因此上述操作对结构类型是以不同的方式实现的。当结构类型的值被转换为 `object` 类型或由该结构实现的接口类型时，就会执行一次装箱操作。与此类似，当 `object` 类型的值或接口类型的值被转换回结构类型时，会执行一次取消装箱操作。与对类类型进行的相同操作相比，主要区别在于：装箱操作会把相关的结构值“复制”到“箱”中，而取消装箱则会从已被装箱的实例中“复制”出一个结构值。因此，在装箱或取消装箱操作后，对“箱”外的结构进行的更改不会影响已被装箱的结构。

有关装箱和取消装箱的详细信息，请参见§4.3。

11.3.6 this 的意义

在类的实例构造函数和实例函数成员中，`this` 为值类别。因此，虽然 `this` 可以用于引用该函数成员调用所涉及的实例，但是不可能在类的函数成员中对 `this` 本身赋值。

在结构的实例构造函数内，`this` 相当于该结构类型的 `out` 参数，而在结构的实例函数成员内，`this` 相当于该结构类型的 `ref` 参数。在这两种情况下，`this` 本身都相当于一个变量，因而有可能对该函数成员调用所涉及的整个结构进行修改（如对 `this` 赋值，或者将 `this` 作为 `ref` 或 `out` 参数传递）。

11.3.7 字段初始值设定项

如§11.3.4 中所描述，结构的默认值就是将所有值类型字段设置为它们的默认值并将所有引用类型字段设置为 `null` 而产生的值。由于这个原因，结构不允许它的实例字段声明中含有变量初始值设定项。此限制只适用于实例字段。在结构的静态字段声明中可以含有

变量初始值设定项。

示例：

```
struct Point
{
    public int x = 1; // 错误，不允许初始值设定项
    public int y = 1; // 错误，不允许初始值设定项
}
```

会出现错误，是因为实例字段声明中含有变量初始值设定项。

11.3.8 构造函数

与类不同，结构不允许声明无参数实例构造函数。实际上，每个结构类型都隐式地含有一个无参数实例构造函数，该构造函数始终返回通过如下方式得到的值：将所有的值类型字段设置为它们的默认值，并将所有引用类型字段设置为 `null` (§4.1.2)。结构可以声明具有参数的实例构造函数。例如：

```
struct Point
{
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

已知以上声明，下列两个语句：

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

都创建了一个 `Point`，而且它们的 `x` 和 `y` 都初始化为零。

结构的实例构造函数不能含有 `base(...)` 形式的构造函数初始值设定项。

如果该结构实例构造函数没有指定构造函数初始值设定项，则 `this` 变量就相当于一个结构类型的 `out` 参数，并且与 `out` 参数类似，`this` 必须在该结构函数返回的每个位置上明确赋值 (§5.3)。如果该结构实例构造函数指定了构造函数初始值设定项，则 `this` 变量就相当于结构类型的 `ref` 参数，并且与 `ref` 参数类似，`this` 被视为在进入构造函数体时已被明确赋值。请研究下面的实例构造函数实现：

```
struct Point
{
    int x, y;
    public int X {
        set { x = value; }
    }
    public int Y {
        set { y = value; }
    }
    public Point(int x, int y) {
        X = x; // 错误，this 还没有被明确赋值
        Y = y; // 错误，this 还没有被明确赋值
    }
}
```

```
    }
}
```

在被构造的结构的所有字段已明确赋值以前，不能调用任何实例成员函数（包括 X 和 Y 属性的 set 访问器）。但是请注意，如果 Point 是类而不是结构，则上述的实例构造函数就变成有效的了。

11.3.9 析构函数

不允许结构声明析构函数。

11.4 结构示例

下面的章节展示了关于应用 struct 类型的两个重要示例，它们各自创建一个类型，这些类型使用起来就像 C# 语言的内置类型，但具有修改了的语义。

11.4.1 数据库整数类型

下面的 DBInt 结构实现了一个整数类型，它可以表示 int 类型的值的完整集合，再加上一个用于表示未知值的附加状态。具有这些特征的类型常用在数据库中。

```
using System;
public struct DBInt
{
    // Null 成员表示为一个未知的 DBInt 值。
    public static readonly DBInt Null = new DBInt();
    // 当字段 defined 为真时，这个 DBInt 表示为已知的值，它被存储在 value 字段中。
    // 当字段 defined 为假时，这个 DBInt 表示为未知的值，同时 value 字段的值为 0。
    int value;
    bool defined;
    // 私有实例构造函数，用一个已知值创建一个 DBInt。
    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }
    // 如果该 DBInt 表示为未知的值，那么，IsNull 属性为真。
    public bool IsNull { get { return !defined; } }
    // Value 属性是这个 DBInt 的已知值；若这个 DBInt 表示未知值，它将是 0。
    public int Value { get { return value; } }
    // 从 int 到 DBInt 的隐式转换
    public static implicit operator DBInt(int x) {
        return new DBInt(x);
    }
    // 从 DBInt 到 int 的显式转换。如果给定的 DBInt 表示未知值，将抛出一个异常。
    public static explicit operator int(DBInt x) {
        if (!x.defined) throw new InvalidOperationException();
        return x.value;
    }
    public static DBInt operator +(DBInt x) {
        return x;
    }
}
```

```

    }
    public static DBInt operator -(DBInt x) {
        return x.defined ? -x.value : Null;
    }
    public static DBInt operator +(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value + y.value: Null;
    }
    public static DBInt operator -(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value - y.value: Null;
    }
    public static DBInt operator *(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value * y.value: Null;
    }
    public static DBInt operator /(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value / y.value: Null;
    }
    public static DBInt operator %(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value % y.value: Null;
    }
    public static DBBool operator ==(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value == y.value: DBBool.Null;
    }
    public static DBBool operator !=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value != y.value: DBBool.Null;
    }
    public static DBBool operator >(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value > y.value: DBBool.Null;
    }
    public static DBBool operator <(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value < y.value: DBBool.Null;
    }
    public static DBBool operator >=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value >= y.value: DBBool.Null;
    }
    public static DBBool operator <=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value <= y.value: DBBool.Null;
    }
    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }
    public override int GetHashCode() {
        return value;
    }
    public override string ToString() {
        return defined? value.ToString(): "DBInt.Null";
    }
}

```

11.4.2 数据库布尔类型

下面的 DBBool 结构实现了一个三值逻辑类型。该类型的可能值有 DBBool.True, DBBool.False 和 DBBool.Null, 其中 Null 成员用于表示未知值。这样的三值逻辑类型经常用在数据库中。

```

using System;
public struct DBBool

```

```

{
    // 3个可能的DBBool值。
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);
    // 私有字段,用-1, 0, 1分别表示False, Null, True。
    sbyte value;
    // 私有实例构造函数,值参数必须是-1、0 或者 1。
    DBBool(int value) {
        this.value = (sbyte)value;
    }
    // 测试DBBool值的属性。如果这个DBBool有给定的值,则返回true;否则,返回false。
    public bool IsNull { get { return value == 0; } }
    public bool IsFalse { get { return value < 0; } }
    public bool IsTrue { get { return value > 0; } }
    // 从bool到DBBool的隐式转换。将true映射到DBBool.True,
    // 并且将false映射到DBBool.False。
    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }
    // 从DBBool到bool的显式转换。如果给定的DBBool为Null,则抛出一个异常;
    // 否则返回true或者false。
    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }
    // 等式运算符。如果任何一个操作数为Null,就返回Null;
    // 否则,返回True或者False。
    public static DBBool operator ==(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value? True: False;
    }
    // 不等式运算符。如果任何一个操作数为Null,就返回Null;
    // 否则,返回True或者False。
    public static DBBool operator !=(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value? True: False;
    }
    // 逻辑非运算符。如果操作数为False,就返回True;如果操作数为Null,
    // 就返回Null;如果操作数为True,就返回False。
    public static DBBool operator !(DBBool x) {
        return new DBBool(-x.value);
    }
    // 逻辑AND运算符。如果任何一个操作数为False,就返回False;
    // 否则,若任何一个操作数为Null,就返回Null;其余情形则返回True。
    public static DBBool operator &(DBBool x, DBBool y) {
        return new DBBool(x.value < y.value? x.value: y.value);
    }
    // 逻辑OR运算符。如果任何一个操作数为True,就返回True;
    // 若任何一个操作数为Null,就返回Null;其余情形则返回False。
    public static DBBool operator |(DBBool x, DBBool y) {
        return new DBBool(x.value > y.value? x.value: y.value);
    }
    // true运算符。如果操作数True,则返回true;否则返回false。
    public static bool operator true(DBBool x) {
        return x.value > 0;
    }
    // false运算符。如果操作数False,则返回true;否则返回false。
    public static bool operator false(DBBool x) {
        return x.value < 0;
    }
}

```

```
}  
public override bool Equals(object obj) {  
    if (!(obj is DBBool)) return false;  
    return value == ((DBBool)obj).value;  
}  
public override int GetHashCode() {  
    return value;  
}  
public override string ToString() {  
    if (value > 0) return "DBBool.True";  
    if (value < 0) return "DBBool.False";  
    return "DBBool.Null";  
}  
}
```

第 12 章 数组

数组是一种包含若干变量的数据结构，这些变量都可以通过计算索引进行访问。数组中包含的变量（又称数组的元素）具有相同的类型，该类型称为数组的元素类型。

数组有一个“秩”，它表示和每个数组元素关联的索引个数。数组的秩又称为数组的维度。秩为 1 的数组称为一维数组（single-dimensional array）。秩大于 1 的数组称为多维数组（multi-dimensional array）。维度大小确定的多维数组通常称为二维数组、三维数组等。

数组的每个维度都有一个关联的长度，它是一个大于或等于零的整数。维度的长度不是数组类型的组成部分，而只与数组类型的实例相关联，它是在运行时创建实例时确定的。维度的长度确定该维度的下标的有效范围：对于长度为 N 的维度，下标范围可以为 0 到 $N-1$ （包括 0 和 $N-1$ ）。数组中的元素总数是数组中各维度长度的乘积。如果数组的一个或多个维度的长度为零，则称该数组为空。

数组的元素类型可以是任意类型，包括数组类型。

12.1 数组类型

数组类型表示为一个非数组类型后跟一个或多个秩说明符：

array-type: (数组类型:)

non-array-type rank-specifiers (非数组类型 秩说明符)

non-array-type: (非数组类型:)

type (类型)

rank-specifiers: (秩说明符:)

rank-specifier (秩说明符)

rank-specifiers rank-specifier (秩说明符 秩说明符)

rank-specifier: (秩说明符:)

[dim-separators_{opt}] ([维度分隔符_{可选}])

dim-separators: (维度分隔符:)

,

dim-separators , (维度分隔符 ,)

上述产生式中，非数组类型是本身不是数组类型的任意类型。

数组类型的秩由数组类型中最左边的秩说明符给定：秩说明符指定该数组的秩等于秩说明符中“,”标记的个数加 1。

数组类型的元素类型就是去掉最左边的秩说明符后剩余表达式的类型：

- 形式为 $T[R]$ 的数组类型是秩为 R ，且元素类型为非数组元素类型 T 的数组。

- 形式为 $T[R][R1]...[RN]$ 的数组类型是秩为 R 、元素类型为 $T[R1]...[RN]$ 的数组。

实质上，在解释数组类型时，先从左到右读取秩说明符，最后才读取那个最终的非数组元素类型。例如，类型 $\text{int}[[,],[,]]$ 表示一个一维数组，该一维数组的元素类型为三维数组，该三维数组的元素类型为二维数组，该二维数组的元素类型为 int 。

在运行时，数组类型的值可以为 `null` 或对该数组类型的某个实例的引用。

`System.Array` 类型是所有数组类型的抽象基类型。存在从任何数组类型到 `System.Array` 的隐式引用转换 (§6.1.4)，并且存在从 `System.Array` 到任何数组类型的显式引用转换 (§6.2.3)。请注意 `System.Array` 本身不是数组类型。相反，它是一个从中派生所有数组类型的类类型。

在运行时，`System.Array` 类型的值可以是 `null` 或对任何数组类型的实例的引用。

12.2 数组创建

数组实例由数组创建表达式 (§7.5.10.2) 创建，或者由包含数组初始值设定项 (§12.6) 的字段声明或局部变量声明创建。

创建数组实例时，将确定秩和各维度的长度，它们在该实例的整个生存期内保持不变。换言之，对于一个已存在的数组实例，既不可能更改它的秩，也不可能调整它的维度大小。

数组实例一定是数组类型。`System.Array` 类型是不能实例化的抽象类型。

由数组创建表达式创建的数组的元素总是被初始化为它们的默认值 (§5.2)。

12.3 数组元素访问

我们可以使用形式为 $A[I1, I2, ..., IN]$ 的“元素访问”表达式 (§7.5.6.1) 访问数组元素，其中 A 是数组类型的表达式，各 IX 是类型为 `int`、`uint`、`long`、`ulong` 的表达式，或者是可隐式转换为这些类型中的一个或多个的表达式。数组元素访问的结果是变量，即由下标选定的数组元素。

此外，还可以使用 `foreach` 语句 (§8.8.4) 来枚举数组的各个元素。

12.4 数组成员

每个数组类型都继承由 `System.Array` 类型所声明的成员。

12.5 数组协方差

对于任意两个引用类型 A 和 B ，如果存在从 A 到 B 的隐式引用转换 (§6.1.4) 或显式引用转换 (§6.2.3)，则也一定存在从数组类型 $A[R]$ 到数组类型 $B[R]$ 的相同的引用

转换，其中 *R* 可以是任何给定的秩说明符，但这两个数组类型必须使用相同的 *R*。这种关系称为数组协方差（array covariance）。具体说来，数组协方差意味着数组类型 *A*[*R*] 的值实际上可能是对数组类型 *B*[*R*] 的实例的引用（如果存在从 *B* 到 *A* 的隐式引用转换的话）。

由于存在数组协方差，因此对引用类型数组的元素的赋值操作会包括一个运行时检查，以确保正在赋给数组元素的值确实是允许的类型（§7.13.1）。例如：

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }
    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

`Fill` 方法中对 `array[i]` 的赋值隐式地包括一个运行时检查，它确保由 `value` 引用的对象是 `null` 或与 `array` 的实际元素类型兼容的类型的实例。在 `Main` 中，`Fill` 的前两个调用成功了，但是在第三个调用中，当执行对 `array[i]` 的第一次赋值时会引发 `System.ArrayTypeMismatchException`。发生此异常是因为装箱的 `int` 类型不能存储在 `string` 数组中。

具体说来，数组协方差不能延伸到“值类型”的数组。例如，不存在允许将 `int[]` 当做 `object[]` 来处理的转换。

12.6 数组初始值设定项

数组初始值设定项可以用于字段声明（§10.4）、局部变量声明（§8.5.1）和数组创建表达式（§7.5.10.2）中：

array-initializer:（数组初始值设定项：）

{ **variable-initializer-list**_{opt} } ({ **变量初始值设定项列表**_{可选} })

{ **variable-initializer-list** , } ({ **变量初始值设定项列表** , })

variable-initializer-list:（变量初始值设定项列表：）

variable-initializer（变量初始值设定项）

variable-initializer-list , **variable-initializer**（变量初始值设定项列表 , 变量初始值设定项）

variable-initializer:（变量初始值设定项：）

expression（表达式）

array-initializer（数组初始值设定项）

数组初始值设定项包含一系列变量初始值设定项，它们括在“{}”标记中并且用“,”标记分隔。每个变量初始值设定项是一个表达式，或者（在多维数组的情况下）是一个嵌

套的数组初始值设定项。

数组初始值设定项所在位置的上下文确定了正在被初始化的数组的类型。在数组创建表达式中，数组类型后面紧跟着初始值设定项。在字段或变量声明中，数组类型就是所声明的字段或变量的类型。当数组初始值设定项用在字段或变量声明中时，如：

```
int[] a = {0, 2, 4, 6, 8};
```

只是下列等效数组创建表达式的简写形式：

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

对于一维数组，数组初始值设定项必须包含一个表达式序列，这些表达式是与数组的元素类型兼容的赋值表达式。这些表达式从下标为零的元素开始，按照升序初始化数组元素。数组初始值设定项中所含的表达式数目确定正在创建的数组实例的长度。例如，上面的数组初始值设定项创建了一个长度为 5 的 `int[]` 实例，并用下列值初始化该实例：

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

对于多维数组，数组初始值设定项必须具有与数组维数相同的嵌套级别。最外面的嵌套级别对应于最左边的维度，而最里面的嵌套级别对应于最右边的维度。数组各维度的长度是由数组初始值设定项中相应嵌套级别内的元素数目确定的。对于每个嵌套的数组初始值设定项，元素的数目必须与同一级别的其他数组初始值设定项所包含的元素数相同。示例：

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

创建一个二维数组，其最左边的维度的长度为 5，最右边的维度的长度为 2：

```
int[,] b = new int[5, 2];
```

然后用下列值初始化该数组实例：

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

当数组创建表达式同时包含显式维度长度和一个数组初始值设定项时，长度必须是常数表达式，并且各嵌套级别的元素数目必须与相应的维度长度匹配。以下是几个示例：

```
int i = 3;
int[] x = new int[3] {0, 1, 2};    // OK
int[] y = new int[i] {0, 1, 2};    // 错误, i 不是一个常量
int[] z = new int[3] {0, 1, 2, 3}; // 错误, 长度/初始值设定项不匹配
```

这里，由于维度长度表达式不是常数，因此 `y` 的初始值设定项导致编译时错误；另外由于初始值设定项中所设定的长度和元素数目不一致，因此 `z` 的初始值设定项也导致编译时错误。

第 13 章 接口

一个接口定义一个约定 (contract)。实现某接口的类或结构必须遵守该接口定义的约定。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。

接口可以包含方法、属性、事件和索引器。接口本身不提供它所定义的成员的实现。接口只指定实现该接口的类或结构必须提供的成员。

13.1 接口声明

接口声明是用于声明新的接口类型的类型声明 (§9.5)。

interface-declaration: (接口声明:)

attributes_{opt} interface-modifiers_{opt} interface identifier interface-base_{opt}
interface-body ;_{opt} (属性_{可选} 接口修饰符_{可选} interface 标识符 接口基_{可选} 接口体 ; _{可选})

接口声明由下列项组成：一个可选的属性集 (§17)，后跟一个可选的接口修饰符集 (§13.1.1)，后面再跟关键字 `interface` 和命名接口的标识符，还可根据需要后跟可选的接口基规范 (§13.1.2)，再跟“接口体” (§13.1.3)，还可选择后跟一个分号。

13.1.1 接口修饰符

接口声明可以根据需要包含一个接口修饰符序列：

interface-modifiers: (接口修饰符:)

interface-modifier (接口修饰符)

interface-modifiers interface-modifier (接口修饰符 接口修饰符)

interface-modifier: (接口修饰符:)

new
public
protected
internal
private

同一修饰符在一个接口声明中出现多次会导致编译时错误。

`new` 修饰符仅允许在类中定义的接口中使用。它指定接口隐藏同名的继承成员，详见 §10.2.2 中的介绍。

`Public`, `protected`, `internal` 和 `private` 修饰符控制接口的可访问性。根据接口声明所在的上下文，只允许使用这些修饰符中的一部分 (§3.5.1)。

13.1.2 基接口

接口可以从零个或多个接口继承，被继承的接口称为该接口的显式基接口（**explicit base interface**）。当接口具有一个或多个显式基接口时，在该接口声明中，接口标识符后就要紧跟一个冒号及一个由逗号分隔的基接口标识符列表。

interface-base:（接口基:）

: **interface-type-list**（: 接口类型列表）

接口的显式基接口必须至少具有与该接口本身相同的可访问性（§3.5.4）。例如，在 **public** 接口的接口基中指定 **private** 或 **internal** 接口就会导致编译时错误。

接口不能从自身直接或间接继承，否则会发生编译时错误。

接口的基接口包括显式基接口，以及这些显式基接口的基接口。换言之，基接口集是显式基接口及它们的显式基接口（依次类推）的完全可传递的闭包。接口继承其基接口的所有成员。对于下面的示例：

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

IComboBox 的基接口是 **IControl**, **ITextBox** 和 **IListBox**。

换言之，上面的 **IComboBox** 接口继承成员 **SetText**，**SetItems** 和 **Paint**。

实现了某接口的类或结构还隐式地实现了该接口的所有基接口。

13.1.3 接口体

接口的接口体定义接口的成员。

interface-body:（接口体:）

{ **interface-member-declarations_{opt}** } ({ 接口成员声明_{可选} })

13.2 接口成员

接口的成员包括从基接口继承的成员和由接口本身声明的成员。

interface-member-declarations:（接口成员声明:）

interface-member-declaration（接口成员声明）

interface-member-declarations interface-member-declaration（接口成员声

明 接口成员声明)

interface-member-declaration: (接口成员声明:)

interface-method-declaration (接口方法声明)

interface-property-declaration (接口属性声明)

interface-event-declaration (接口事件声明)

interface-indexer-declaration (接口索引器声明)

接口声明可以声明零个或多个成员。接口的成员必须是方法、属性、事件或索引器。接口不能包含常数、字段、运算符、实例构造函数、析构函数或类型，也不能包含任何种类的静态成员。

所有接口成员都隐式地具有 `public` 访问属性。接口成员声明中包含任何修饰符都属于编译时错误。具体来说，不能用修饰符 `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override` 或 `static` 来声明接口成员。

示例:

```
public delegate void StringListEvent(IStringList sender);
public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

声明了一个接口，该接口的成员涵盖了所有可能作为接口成员的种类（每种一个）：方法、属性、事件和索引器。

接口声明创建新的声明空间 (§3.3)，并且接口声明直接包含的接口成员声明将新成员提供给该声明空间。以下规则适用于接口成员声明：

- 方法的名称必须与同一接口中声明的所有属性和事件的名称不同。此外，方法的签名 (§3.6) 必须与同一接口中声明的所有其他方法的签名不同。
- 属性或事件的名称必须与同一接口中声明的所有其他成员的名称不同。
- 索引器的签名必须区别于在同一接口中声明的其他所有索引器的签名。

准确地说，接口所继承的成员不是该接口的声明空间的一部分。因此，允许接口用与它所继承的成员相同的名称或签名来声明新的成员。发生这种情况时，则称派生的接口成员隐藏了基接口成员。隐藏继承的成员不算是错误，但这确实会导致编译器发出警告。为了避免出现上述警告，派生接口成员的声明中必须包含一个 `new` 修饰符，以指示该派生成员将要隐藏对应的基成员。§3.7.2 中对该主题进行了进一步讨论。

如果在不隐藏所继承成员的声明中包含 `new` 修饰符，则会对此发出警告。通过移除 `new` 修饰符可避免出现此警告。

13.2.1 接口方法

接口方法是使用“接口方法声明”来声明的：

interface-method-declaration: (接口方法声明:)

`attributesopt newopt return-type identifier (formal-parameter-listopt);` (属性可选 `new` 可选 返回类型 标识符 (形参表可选);)

接口方法声明中的“属性”、“返回类型”、“标识符”和“形参表”与类中的方法声明的对应项 (§10.5) 具有相同的意义。不允许接口方法声明指定方法体，因此，声明总是以分号结尾。

13.2.2 接口属性

接口属性是使用“接口属性声明”来声明的：

`interface-property-declaration:` (接口属性声明:)

`attributesopt newopt type identifier { interface-accessors } (属性可选 new可选 类型 标识符 { 接口访问器 })`

`interface-accessors:` (接口访问器:)

`attributesopt get ; (属性可选 get ;)`

`attributesopt set ; (属性可选 set ;)`

`attributesopt get ; attributesopt set ; (属性可选 get ; 属性可选 set ;)`

`attributesopt set ; attributesopt get ; (属性可选 set ; 属性可选 get ;)`

接口属性声明中的“属性”、“类型”和“标识符”与类中的属性声明的对应项 (§10.6) 具有相同的意义。

接口属性声明的访问器与类属性声明 (§10.6.2) 的访问器相对应，不同之处在于接口属性声明的访问器体必须始终是一个分号。因此，访问器在这里只用于表示该属性为读写、只读还是只写。

13.2.3 接口事件

接口事件是使用“接口事件声明”来声明的：

`interface-event-declaration:` (接口事件声明:)

`attributesopt newopt event type identifier; (属性可选 new可选 event 类型 标识符 ;)`

接口事件声明中的“属性”、“类型”和“标识符”与类中事件声明的对应项 (§10.7) 具有相同的意义。

13.2.4 接口索引器

接口索引器是使用“接口索引器声明”来声明的：

`interface-indexer-declaration:` (接口索引器声明:)

`attributesopt newopt type this [formal-parameter-list] { interface-accessors } (属性可选 new可选 类型 this [形参表] { 接口访问器 })`

接口索引器声明中的“属性”、“类型”和“形参表”与类中索引器声明的对应项

(§10.8) 具有相同的意义。

接口索引器声明的访问器与类索引器声明 (§10.8) 的访问器相对应, 不同之处在于接口索引器声明的访问体必须始终是一个分号。因此, 访问器在这里只用于表示该索引器为读写、只读还是只写。

13.2.5 接口成员访问

接口成员是通过 `IM` 形式的成员访问 (§7.5.4) 表达式和 `I[A]` 形式的索引器访问 (§7.5.6.2) 表达式来访问的, 其中 `I` 是接口类型, `M` 是该接口类型的方法、属性或事件, `A` 是对应的索引器参数列表。

对于严格单一继承 (继承链中的每个接口均恰好有零个或一个直接基接口) 的接口, 成员查找 (§7.3)、方法调用 (§7.5.5.1) 和索引器访问 (§7.5.6.2) 规则的效果与类和结构的完全相同: 派生程度较大的成员隐藏具有相同名称或签名的派生程度较小的成员。然而, 对于多重继承接口, 当两个或更多个不相关 (互不继承) 的基接口中声明了具有相同名称或签名的成员时, 就会发生多义性。本节列出了此类情况的几个示例。在所有情况下, 都可以使用显式强制转换来解决这种多义性。

对于下面的示例:

```
interface IList
{
    int Count { get; set; }
}
interface ICounter
{
    void Count(int i);
}
interface IListCounter: IList, ICounter {}
class C
{
    void Test(IListCounter x) {
        x.Count(1);           // 错误
        x.Count = 1;          // 错误
        ((IList)x).Count = 1;  // Ok, 调用 IList.Count.set
        ((ICounter)x).Count(1); // Ok, 调用 ICounter.Count
    }
}
```

由于在 `IListCounter` 中对 `Count` 的成员查找 (§7.3) 所获得的结果是不明确的, 因此前两个语句将导致编译时错误。如示例所阐释的, 将 `x` 强制转换为适当的基接口类型就可以解决这种多义性。此类强制转换没有运行时开销, 它们只是在编译时将该实例所属类型视为派生程度较小的类型而已。

对于下面的示例:

```
interface IInteger
{
    void Add(int i);
}
interface IDouble
{
```



```

    void Add(double d);
}
interface INumber: IInteger, IDouble {}
class C
{
    void Test(INumber n) {
        n.Add(1);           // 错误, 这两个方法都适合
        n.Add(1.0);         // Ok, 只有 IDouble.Add 是适合的
        ((IInteger)n).Add(1); // Ok, 只有 IInteger.Add 是候选者
        ((IDouble)n).Add(1);  // Ok, 只有 IDouble.Add 是候选者
    }
}

```

由于方法调用 (§7.5.5.1) 要求以相同类型声明所有的重载候选方法, 因此调用 `n.Add(1)` 是不明确的。然而, 调用 `n.Add(1.0)` 是允许的, 这是因为只有 `IDouble.Add` 适用。插入显式强制转换后, 就只有一个候选方法了, 因此没有多义性。

对于下面的示例:

```

interface IBase
{
    void F(int i);
}
interface ILeft: IBase
{
    new void F(int i);
}
interface IRight: IBase
{
    void G();
}
interface IDerived: ILeft, IRight {}
class A
{
    void Test(IDerived d) {
        d.F(1);           // 调用 ILeft.F
        ((IBase)d).F(1);   // 调用 IBase.F
        ((ILeft)d).F(1);   // 调用 ILeft.F
        ((IRight)d).F(1);  // 调用 IBase.F
    }
}

```

`IBase.F` 成员被 `ILeft.F` 成员隐藏。因此, 即使在通过 `IRight` 的访问路径中 `IBase.F` 似乎没有被隐藏, 调用 `d.F(1)` 仍选择 `ILeft.F`。

多重继承接口中隐藏的直观规则简单地说就是: 如果成员在任何一个访问路径中被隐藏, 那么它在所有访问路径中都被隐藏。由于从 `IDerived` 经 `ILeft` 到 `IBase` 的访问路径隐藏了 `IBase.F`, 因此该成员在从 `IDerived` 经 `IRight` 到 `IBase` 的访问路径中也被隐藏。

13.3 完全限定接口成员名

接口成员有时也用它的完全限定名 (fully qualified name) 来引用。接口成员的完全限定名是这样组成的: 声明该成员的接口的名称, 后跟一个点, 再跟该成员的名称。成员的完全限定名将引用声明该成员的接口。例如, 给定下列声明:

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
```

`Paint` 的完全限定名是 `IControl.Paint`，而 `SetText` 的完全限定名是 `ITextBox.SetText`。在上面的示例中，不能用 `ITextBox.Paint` 来引用 `Paint` 方法。

当接口是命名空间的组成部分时，该接口成员的完全限定名需包含命名空间名称。例如：

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

这里，`Clone` 方法的完全限定名是 `System.ICloneable.Clone`。

13.4 接口实现

接口可以由类和结构来实现。为了标识类或结构实现了某接口，在该类或结构的基类列表中应该包含该接口的标识符。例如：

```
interface ICloneable
{
    object Clone();
}
interface IComparable
{
    int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

如果类或结构实现某个接口，则它还隐式地实现该接口的所有基接口。即使在类或结构的基类列表中没有显式地列出所有基接口，也是如此。例如：

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
class TextBox: ITextBox
```

```

{
    public void Paint() {...}
    public void SetText(string text) {...}
}

```

这里，`TextBox` 类同时实现了 `IControl` 和 `ITextBox`。

13.4.1 显式接口成员实现

为了实现接口，类或结构可以声明显式接口成员实现（explicit interface member implementations）。显式接口成员实现就是一种方法、属性、事件或索引器声明，它使用完全限定接口成员名称作为标识符。例如：

```

interface ICloneable
{
    object Clone();
}
interface IComparable
{
    int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}
}

```

这里，对 `ICloneable.Clone` 和 `IComparable.CompareTo` 的声明是显式接口成员实现。

某些情况下，接口成员的名称对于实现该接口的类可能是不适当的，此时，可以使用显式接口成员实现来实现该接口成员。例如，一个用于文件抽象的类一般会实现一个具有释放文件资源作用的 `Close` 成员函数，还可能使用显式接口成员实现来实现 `IDisposable` 接口的 `Dispose` 方法：

```

interface IDisposable
{
    void Dispose();
}
class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }
    public void Close() {
        // 关闭该文件
        System.GC.SuppressFinalize(this);
    }
}

```

在方法调用、属性访问或索引器访问中，不能直接访问显式接口成员实现的成员，即使用它的完全限定名也不行。显式接口成员实现的成员只能通过接口实例访问，并且在通过接口实例访问时，只能用该接口成员的简单名称来引用。

显式接口成员实现中包含访问修饰符会导致编译时错误，而且如果包含 `abstract`，

virtual, override 或 static 修饰符也会导致编译时错误。

显式接口成员实现具有与其他成员不同的可访问性特征。由于显式接口成员实现永远不能在方法调用或属性访问中通过它们的完全限定名来访问，因此，它们似乎是 private（私有的）。但是，因为它们可以通过接口实例来访问，所以它们似乎又是 public（公有的）。

显式接口成员实现有两个主要用途：

- 由于显式接口成员实现不能通过类或结构实例来访问，因此它们就不属于类或结构的自身的公共接口。当需要在一个公用的类或结构中实现一些仅供内部使用（不允许外界访问）的接口时，这特别有用。
- 显式接口成员实现可以消除因同时含有多个相同签名的接口成员所引起的多义性。如果没有显式接口成员实现，类或结构就不可能为具有相同签名和返回类型的接口成员分别提供相应的实现，也不可能为具有相同签名和不同返回类型的所有接口成员中的任何一个提供实现。

为了使显式接口成员实现有效，声明它的类或结构必须在它的基类列表中指定一个接口，而该接口必须包含一个成员，该成员的完全限定名、类型和参数类型与该显式接口成员实现所具有的完全相同。因此，对于下列类：

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...} // 无效
}
```

IComparable.CompareTo 声明将导致编译时错误，原因是 **IComparable** 未列在 **Shape** 的基类列表中，并且不是 **ICloneable** 的基接口。与此类似，对于下列声明：

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}
class Ellipse: Shape
{
    object ICloneable.Clone() {...} // 无效
}
```

Ellipse 中的 **ICloneable.Clone** 声明也将导致编译时错误，因为 **ICloneable** 未在 **Ellipse** 的基类列表中显式列出。

接口成员的完全限定名必须引用声明该成员的接口。因此，对于下列声明：

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

Paint 的显式接口成员实现必须写为 `IControl.Paint`。

13.4.2 接口映射

类或结构必须为它的基类列表中所列出的接口的所有成员提供它自己的实现。在进行实现的类或结构中定位接口成员的实现的过程称为接口映射。

关于类或结构 `C` 的接口映射就是查找 `C` 的基类列表中指定的每个接口的每个成员的实现。对某个特定接口成员 `I.M` 的实现（其中 `I` 是声明了成员 `M` 的接口）的定位按下述规则执行：从 `C` 开始，按继承顺序，逐个检查它的每个后续基类（下面用 `S` 表示每个进行检查的类或结构），直到找到匹配项。

- 如果 `S` 包含一个与 `I` 和 `M` 匹配的显式接口成员实现的声明，那么此成员就是 `I.M` 的实现。
- 如果 `S` 包含与 `M` 匹配的非静态的 `public` 成员声明，则此成员就是 `I.M` 的实现。

如果不能为 `C` 的基类列表中指定的所有接口的所有成员找到实现，则将发生编译时错误。请注意，接口的成员包括那些从基接口继承的成员。

根据接口映射的含义，在下列情况下类成员 `A` 与接口成员 `B` 匹配：

- `A` 和 `B` 都是方法，并且 `A` 和 `B` 的名称、类型和形参表都相同。
- `A` 和 `B` 都是属性，`A` 和 `B` 的名称与类型相同，并且 `A` 与 `B` 具有相同的访问器（如果 `A` 不是显式接口成员实现，则它可以具有其他访问器）。
- `A` 和 `B` 都是事件，并且 `A` 和 `B` 的名称与类型相同。
- `A` 和 `B` 都是索引器，`A` 和 `B` 的类型与形参表相同，并且 `A` 与 `B` 具有相同的访问器（如果 `A` 不是显式接口成员实现，则它可以具有其他访问器）。

接口映射算法中隐含着下列值得注意的特征：

- 在类或结构成员中确定哪个实现了接口成员时，显式接口成员实现比同一个类或结构中的其他成员具有更高的优先级。
- 接口映射不涉及非公共成员和静态成员。

对于下面的示例：

```
interface ICloneable
{
    object Clone();
}
class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

`C` 的 `ICloneable.Clone` 成员成为 `ICloneable` 中 `Clone` 的实现，这是因为显式接口成员实现比其他成员具有更高的优先级。

如果类或结构实现两个或更多个接口，而这些接口包含具有相同名称、类型和参数类型的成员，则这些接口成员可以全部映射到单个类或结构成员上。例如：

```
interface IControl
{
    void Paint();
}
interface IForm
{
    void Paint();
}
class Page: IControl, IForm
{
    public void Paint() {...}
}
```

此处，IControl 和 IForm 的 Paint 方法都被映射到 Page 中的 Paint 方法上。当然也可以为这两个方法提供单独的显式接口成员实现。

如果类或结构实现一个包含被隐藏成员的接口，那么某些成员必须通过显式接口成员实现来实现。例如：

```
interface IBase
{
    int P { get; }
}
interface IDerived: IBase
{
    new int P();
}
```

实现接口将至少需要一个显式接口成员实现，可采取下列形式之一：

```
class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}
class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}
class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}
```

当一个类实现多个具有相同基接口的接口时，为该基接口提供的实现只能有一个。对于下面的示例：

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{

```

```

        void SetItems(string[] items);
    }
    class ComboBox: IControl, ITextBox, IListBox
    {
        void IControl.Paint() {...}
        void ITextBox.SetText(string text) {...}
        void IListBox.SetItems(string[] items) {...}
    }

```

在基类列表中命名的 `IControl`、由 `ITextBox` 继承的 `IControl` 和由 `IListBox` 继承的 `IControl` 不可能有各自不同的实现。事实上，没有为这些接口提供单独实现的打算。相反，`ITextBox` 和 `IListBox` 的实现共享相同的 `IControl` 的实现，因而可以简单地认为 `ComboBox` 实现了三个接口：`IControl`，`ITextBox` 和 `IListBox`。

基类的成员参与接口映射。对于下面的示例：

```

interface Interface1
{
    void F();
}
class Class1
{
    public void F() {}
    public void G() {}
}
class Class2: Class1, Interface1
{
    new public void G() {}
}

```

`Class1` 中的方法 `F` 被当做 `Class2` 为 `Interface1` 提供的实现。

13.4.3 接口实现继承

类继承由其基类提供的所有接口实现。

如果不显式地重新实现接口，派生类就无法以任何方式更改它从其基类继承的接口映射。例如，对于下面的声明：

```

interface IControl
{
    void Paint();
}
class Control: IControl
{
    public void Paint() {...}
}
class TextBox: Control
{
    new public void Paint() {...}
}

```

`TextBox` 中的 `Paint` 方法隐藏 `Control` 中的 `Paint` 方法，但这种隐藏并不更改 `Control.Paint` 到 `IControl.Paint` 的映射，所以通过类实例和接口实例对 `Paint` 进行的调用就将具有不同的结果：


```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();           // 调用 Control.Paint();
t.Paint();           // 调用 TextBox.Paint();
ic.Paint();           // 调用 Control.Paint();
it.Paint();           // 调用 Control.Paint();
```

但是，当接口方法被映射到类中的虚拟方法上时，从该类派生的类若重写了该虚拟方法，则将同时更改该接口的实现。例如，将上面的声明改写为：

```
interface IControl
{
    void Paint();
}
class Control: IControl
{
    public virtual void Paint() {...}
}
class TextBox: Control
{
    public override void Paint() {...}
}
```

将产生下列效果：

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();           // 调用 Control.Paint();
t.Paint();           // 调用 TextBox.Paint();
ic.Paint();           // 调用 Control.Paint();
it.Paint();           // 调用 TextBox.Paint();
```

由于显式接口成员实现不能被声明为虚拟的，因此不可能重写显式接口成员实现。然而，显式接口成员实现完全可以调用另一个方法，只要将该方法声明为虚拟方法，派生类就可以重写它了。例如：

```
interface IControl
{
    void Paint();
}
class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}
class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

这里，从 **Control** 派生的类可以通过重写 **PaintControl** 方法来专用化 **IControl.Paint** 的实现。

13.4.4 接口重新实现

一个类若继承了某个接口的实现，则只要将该接口列入它的基类列表中，就可以重新实现该接口。

接口的重新实现与接口的初始实现遵循完全相同的接口映射规则。因此，继承的接口映射不会对为重新实现该接口而建立的接口映射产生任何影响。例如，对于下面的声明：

```
interface IControl
{
    void Paint();
}
class Control: IControl
{
    void IControl.Paint() {...}
}
class MyControl: Control, IControl
{
    public void Paint() {}
}
```

其中，Control 将 IControl.Paint 映射到 Control.IControl.Paint 上，而这并不会影响 MyControl 中 IControl.Paint 的重新实现，在此重新实现中，会将 IControl.Paint 映射到 MyControl.Paint 上。

被继承的公共成员声明和被继承的显式接口成员声明可以参与重新实现接口的接口映射过程。例如：

```
interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}
class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}
class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}
```

此处，Derived 中 IMethods 的实现将各个接口方法分别映射到 Derived.F，Base.IMethods.G，Derived.IMethods.H 和 Base.I 上。

当类实现接口时，它还隐式地实现该接口的所有基接口。与此类似，接口的重新实现也同时隐式地对该接口的所有基接口进行重新实现。例如：

```
interface IBase
{
    ...
}
```

```

        void F();
    }
    interface IDerived: IBase
    {
        void G();
    }
    class C: IDerived
    {
        void IBase.F() {...}
        void IDerived.G() {...}
    }
    class D: C, IDerived
    {
        public void F() {...}
        public void G() {...}
    }

```

这里，IDerived 的重新实现重新实现了 IBase，并将 IBase.F 映射到 D.F 上。

13.4.5 抽象类和接口

与非抽象类类似，抽象类也必须为在该类的基类列表中所列出的接口的所有成员提供它自己的实现。但是，允许抽象类将接口方法映射到抽象方法上。例如：

```

interface IMethods
{
    void F();
    void G();
}
abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}

```

这里，IMethods 的实现将 F 和 G 映射到抽象方法上，这些抽象方法必须在从 C 派生的非抽象类中重写。

注意，显式接口成员实现本身不能是抽象的，但是当然允许显式接口成员实现调用抽象方法。例如：

```

interface IMethods
{
    void F();
    void G();
}
abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}

```

这里，从 C 派生的非抽象类被要求重写 FF 和 GG，从而提供 IMethods 的实际实现。

第 14 章 枚举

枚举类型是一种独特的值类型 (§4.1)，它用于声明一组命名的常数。

示例：

```
enum Color
{
    Red,
    Green,
    Blue
}
```

声明一个名为 Color 的枚举类型，它具有三个成员：Red，Green 和 Blue。

14.1 枚举声明

枚举声明用于声明新的枚举类型。枚举声明以关键字 enum 开始，然后定义该枚举的名称、可访问性、基础类型和成员。

```
enum-declaration: (枚举声明:)
    attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt
(属性可选 枚举修饰符可选 enum 标识符 枚举基可选 枚举体 ;可选)
enum-base: (枚举基:)
    : integral-type (: 整型)
enum-body: (枚举体:)
    { enum-member-declarationsopt } ({ 枚举成员声明可选 })
    { enum-member-declarations , } ({ 枚举成员声明 , })
```

每个枚举类型都有一个相应的整型，称为该枚举类型的基础类型 (underlying type)。此基础类型必须能够表示该枚举中定义的所有枚举数值。枚举声明可以显式地声明 byte，sbyte，short，ushort，int，uint，long 或 ulong 类型作为对应的基础类型。请注意 char 不能用做基础类型。没有显式地声明基础类型的枚举声明意味着所对应的基础类型是 int。

示例：

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

声明了一个基础类型为 long 的枚举。开发人员可以像本示例一样选择使用 long 基础类型，以便能够使用在 long 范围内而不是在 int 范围内的值，或者保留此选项供将来使用。

14.2 枚举修饰符

枚举声明可以根据需要包含一个枚举修饰符序列：

enum-modifiers: (枚举修饰符:)

enum-modifier (枚举修饰符)

enum-modifiers enum-modifier (枚举修饰符 枚举修饰符)

enum-modifier: (枚举修饰符:)

```
new
public
protected
internal
private
```

同一修饰符在一个枚举声明中多次出现会导致编译时错误。

枚举声明的修饰符与类声明的修饰符 (§10.1.1) 具有同样的意义。然而，请注意，在枚举声明中不允许使用 `abstract` 修饰符和 `sealed` 修饰符。枚举不能是抽象的，也不允许派生。

14.3 枚举成员

枚举类型声明的体用于定义零个或多个枚举成员，这些成员是该枚举类型的命名常数。任意两个枚举成员不能具有相同的名称。

enum-member-declarations: (枚举成员声明:)

enum-member-declaration (枚举成员声明)

enum-member-declarations , enum-member-declaration (枚举成员声明 , 枚举成员声明)

enum-member-declaration: (枚举成员声明:)

attributes_{opt} identifier (属性_{可选} 标识符)

attributes_{opt} identifier = constant-expression (属性_{可选} 标识符 = 常数表达式)

每个枚举成员均具有相关联的常数值。此值的类型就是包含了它的那个枚举的基础类型。每个枚举成员的常数值必须在该枚举的基础类型的范围之内。示例：

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

将导致编译时错误，原因是常数值 `-1`、`-2` 和 `-3` 不在基础整型 `uint` 的范围内。

多个枚举成员可以共享同一个关联值。示例：

```
enum Color
{
    Red,
```

```

    Green,
    Blue,
    Max = Blue
}

```

显示一个枚举，其中的两个枚举成员（Blue 和 Max）具有相同的关联值。

一个枚举成员的关联值或隐式地、或显式地被赋值。如果枚举成员的声明中具有常数表达式初始值设定项，则该常数表达式的值（它隐式地转换为枚举的基础类型）就是该枚举成员的关联值。如果枚举成员的声明不具有初始值设定项，则按下面规则隐式地设置它的关联值：

- 如果枚举成员是在枚举类型中声明的第一个枚举成员，则它的关联值为零。
- 否则，枚举成员的关联值是通过将前一个枚举成员（按照文本顺序）的关联值加 1 得到的。这样增加后的值必须在该基础类型可表示的值的范围内；否则，会出现编译时错误。

示例：

```

using System;
enum Color
{
    Red,
    Green = 10,
    Blue
}
class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }
    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);
            case Color.Green:
                return String.Format("Green = {0}", (int) c);
            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);
            default:
                return "Invalid color";
        }
    }
}

```

输出枚举成员名称和它们的关联值。输出为：

```

Red = 0
Green = 10
Blue = 11

```

原因如下：

- 枚举成员 Red 被自动赋予值零（因为它不具有初始值设定项并且是第一个枚举成员）。
- 枚举成员 Green 被显式赋予值 10。

- 枚举成员 `Blue` 被自动赋予比文本上位于它前面的成员大 1 的值。

枚举成员的关联值不能直接或间接地使用它自己的关联枚举成员的值。除了这个循环性限制外，枚举成员初始值设定项可以自由地引用其他的枚举成员初始值设定项，而不必考虑它们所在的文本位置的排列顺序。在枚举成员初始值设定项内，其他枚举成员的值始终被视为属于所对应的基础类型，因此在引用其他枚举成员时，没有必要使用强制转换。

示例：

```
enum Circular
{
    A = B,
    B
}
```

产生编译时错误，因为 `A` 和 `B` 的声明是循环的。`A` 显式依赖于 `B`，而 `B` 隐式依赖于 `A`。

枚举成员的命名方式和作用范围与类中的字段完全类似。枚举成员的范围是包含了它的枚举类型的体。在该范围内，枚举成员可以用它们的简单名称引用。在所有其他代码中，枚举成员的名称必须用它的枚举类型的名称限定。枚举成员不具有任何声明可访问性，如果一个枚举类型是可访问的，则它所含的所有枚举成员都是可访问的。

14.4 System.Enum 类型

`System.Enum` 类型是所有枚举类型的抽象基类（它是一种与枚举类型的基础类型不同的独特类型），并且从 `System.Enum` 继承的成员在任何枚举类型中都可用。存在从任何枚举类型到 `System.Enum` 的装箱转换 (§4.3.1)，并且存在从 `System.Enum` 到任何枚举类型的取消装箱转换 (§4.3.2)。

请注意 `System.Enum` 本身不是枚举类型。相反，它是一个类类型，所有枚举类型都是从它派生的。类型 `System.Enum` 从类型 `System.ValueType` (§4.1.1) 派生，而后者又从类型 `object` 派生。在运行时，类型 `System.Enum` 的值可以是 `null` 或是对任何枚举类型的装了箱的值的引用。

14.5 枚举值和运算

每个枚举类型都定义了一个独特类型；需要使用显式枚举转换 (§6.2.2) 在枚举类型和整型之间或在两个枚举类型之间进行转换。枚举类型的值域不受它的枚举成员限制。具体说来，枚举的基础类型的任何一个值都可以被强制转换为该枚举类型，成为该枚举类型的一个独特的有效值。

枚举成员所属的类型就是声明了它们的那个枚举类型（出现在其他枚举成员初始值设定项中时除外，请参见§14.3）。在枚举类型 `E` 中声明且关联值为 `v` 的枚举成员的值为 `(E)v`。

下列运算符可以用在枚举类型的值上：`==`，`!=`，`<`，`>`，`<=`，`>=` (§7.9.5)，二元`+` (§7.7.4)，二元`-` (§7.7.5)，`^`，`&`，`|` (§7.10.2)，`~` (§7.6.4)，`++`、`--` (§7.5.9 和§7.6.5)、`sizeof` (§18.5.4)。

第 15 章 委托

委托是用来处理其他语言（如 C++，Pascal 和 Modula）需用函数指针来处理的情况的。不过与 C++ 函数指针不同，委托是完全面对对象的；另外，C++ 指针仅指向成员函数，而委托同时封装了对象实例和方法。

委托声明定义一个从 `System.Delegate` 类派生的类。委托实例封装一个调用列表，该列表列出一个或多个方法，其中每个方法均作为一个可调用实体来引用。对于实例方法，可调用实体由该方法和一个相关联的实例组成。对于静态方法，可调用实体仅由一个方法组成。用一个适当的参数集来调用一个委托实例，就是用此给定的参数集来调用该委托实例的每个可调用实体。

委托实例的一个有趣且有用的属性是：它不知道也不关心它所封装的方法所属的类；它所关心的仅限于这些方法必须与委托的类型兼容 (§15.1)。这使委托非常适合于“匿名”调用。

15.1 委托声明

委托声明是一种类型声明 (§9.5)，它声明一个新的委托类型。

delegate-declaration: (委托声明:)

```
attributesopt delegate-modifiersopt delegate  
return-type identifier ( formal-parameter-listopt ); ( 属性 可选 委托修饰符 可选  
delegate 返回类型 标识符 ( 形参表可选 ) ;)
```

delegate-modifiers: (委托修饰符:)

delegate-modifier (委托修饰符)

delegate-modifiers delegate-modifier (委托修饰符 委托修饰符)

delegate-modifier: (委托修饰符:)

```
new  
public  
protected  
internal  
private
```

同一修饰符在一个委托声明中多次出现会导致编译时错误。

`new` 修饰符仅允许在其他类型中声明的委托上使用，在这种情况下该修饰符表示所声明的委托会隐藏具有相同名称的继承成员，详见 §10.2.2。

`public`, `protected`, `internal` 和 `private` 修饰符控制委托类型的可访问性。根据委托声明所在的上下文，可能不允许使用其中某些修饰符 (§3.5.1)。

上述的语法产生式中，标识符用于指定委托的类型名称。

可选的形参表用于指定委托的参数，而返回类型则指定委托的返回类型。如果下面两个条件都为真，则方法和委托类型是兼容的（compatible）：

- 它们具有相同的参数数目，并且类型相同，顺序相同，参数修饰符也相同。
- 它们的返回类型相同。

C# 中的委托类型是名称等效的，而不是结构等效的。具体地说，对于两个委托类型，即使它们具有相同的参数列表和返回类型，仍被认为是不同的两个委托类型。不过，这样两个彼此不同但结构上又相同的委托类型，它们的实例在比较时可以认为是相等关系（§7.9.8）。

例如：

```
delegate int D1(int i, double d);
class A
{
    public static int M1(int a, double b) {...}
}
class B
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

其中，委托类型 D1 和 D2 都与方法 A.M1 和 B.M1 兼容，这是因为它们具有相同的返回类型和参数列表；但是，这些委托类型是两个不同的类型，所以它们是不可互换的。委托类型 D1 和 D2 与方法 B.M2、B.M3 和 B.M4 不兼容，这是因为它们具有不同的返回类型或参数列表。

声明委托类型的惟一方法是通过委托声明。委托类型是从 System.Delegate 派生的类类型。委托类型隐含为 sealed，所以不允许从一个委托类型派生任何类型。也不允许从 System.Delegate 派生非委托类类型。请注意：System.Delegate 本身不是委托类型；它是从中派生所有委托类型的类类型。

C# 设置了专门的语法用于委托类型的实例化和调用。除实例化外，所有可以应用于类或类实例的操作也可以相应地应用于委托类或委托实例。具体说来，可以通过通常的成员访问语法访问 System.Delegate 类型的成员。

委托实例所封装的方法集合称为调用列表。从某个方法创建一个委托实例时（§15.2），该委托实例将封装此方法，此时，它的调用列表只包含一个进入点。但是，当组合两个非空委托实例时，它们的调用列表被连接在一起（按照左操作数在先、右操作数在后的顺序）以组成一个新的调用列表，它包含了两个或更多个进入点。

委托是使用二元 +（§7.7.4）和 += 运算符（§7.13.2）来组合的。可以使用一元 -（§7.7.5）和 -= 运算符（§7.13.2）将一个委托从委托组合中移除。委托间还可以进行比较以确定它们是否相等（§7.9.8）。

下面的示例显示多个委托的实例化及其相应的调用列表：

```

delegate void D(int x);
class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}
}
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);      // M1
        D cd2 = new D(C.M2);      // M2
        D cd3 = cd1 + cd2;        // M1 + M2
        D cd4 = cd3 + cd1;        // M1 + M2 + M1
        D cd5 = cd4 + cd3;        // M1 + M2 + M1 + M1 + M2
    }
}

```

实例化 `cd1` 和 `cd2` 时，它们分别封装一个方法。实例化 `cd3` 时，它的调用列表有两个方法 `M1` 和 `M2`，而且正是这样的顺序。`cd4` 的调用列表包含 `M1`，`M2` 和 `M1`，顺序与此相同。最后，`cd5` 的调用列表包含：`M1`，`M2`，`M1`，`M1` 和 `M2`，顺序与此相同。有关组合（以及移除）委托的更多示例，请参见§15.3。

15.2 委托实例化

委托的实例是由“委托创建表达式” (§7.5.10.3) 创建的。因此，新创建的委托实例将引用以下各项之一：

- 委托创建表达式中引用的静态方法。
- 委托创建表达式中引用的目标对象（此对象不能为 `null`）和实例方法。
- 另一个委托。

例如：

```

delegate void D(int x);
class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);      // 静态方法
        Test t = new C();
        D cd2 = new D(t.M2);      // 实例方法
        D cd3 = new D(cd2);      // 另一个委托
    }
}

```

委托实例一旦被实例化，将始终引用同一目标对象和方法。记住，当组合两个委托或者从一个委托移除另一个时，将产生一个新的委托，该委托具有它自己的调用列表；被组合或移除的委托的调用列表将保持不变。

15.3 委托调用

C# 为调用委托提供了专门的语法。当调用非空的、调用列表仅包含一个进入点的委托实例时，它调用调用列表中的方法，委托调用所使用的参数和返回的值均与该方法的对应项相同。有关委托调用的详细信息，请参见§7.5.5.2。如果在对这样的委托进行调用期间发生异常，而且没有在被调用的方法内捕捉到该异常，则会在调用该委托的方法内继续搜索与该异常对应的 `catch` 子句，就像调用该委托的方法直接调用了该委托所引用的方法一样。

如果一个委托实例的调用列表包含多个进入点，那么调用这样的委托实例就是按顺序同步地调用调用列表中所列的各个方法。以这种方式调用的每个方法都使用相同的参数集，即提供给委托实例的参数集。如果这样的委托调用包含引用参数 (§10.5.1.2)，那么每个方法调用都将使用对同一变量的引用；这样，若调用列表中有某个方法对该变量进行了更改，则调用列表中排在该方法之后的所有方法都会见到此变更。如果委托调用包含输出参数或一个返回值，则它们的最终值就是调用列表中最后一个方法调用所产生的结果。

如果在处理此类委托的调用期间发生异常，而且没有在正被调用的方法内捕捉到该异常，则会在调用该委托的方法内继续搜索与该异常对应的 `catch` 子句，此时，调用列表中排在后面的任何方法将不会被调用。

试图调用其值为 `null` 的委托实例将导致 `System.NullReferenceException` 类型的异常。

下列示例展示如何实例化、组合、移除和调用委托：

```
using System;
delegate void D(int x);
class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }
    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }
    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);                // 调用 M1
        D cd2 = new D(C.M2);
        cd2(-2);                // 调用 M2
        D cd3 = cd1 + cd2;
        cd3(10);                // 调用 M1, 然后调用 M2
        cd3 += cd1;
        cd3(20);                // 调用 M1, M2, 然后调用 M1
        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
```

```

        cd3(30);           // 调用 M1, M2, M1, 然后调用 M3
        cd3 -= cd1;        // 移除最后一个 M1
        cd3(40);          // 调用 M1, M2, 然后调用 M3
        cd3 -= cd4;
        cd3(50);           // 调用 M1 然后调用 M2
        cd3 -= cd2;
        cd3(60);           // 调用 M1
        cd3 -= cd2;        // 不可能的移除不算是错误
        cd3(60);           // 调用 M1
        cd3 -= cd1;        // 调用列表是空的, 因此 cd3 为 null
        //      cd3(70);    // 抛出 System.NullReferenceException
        cd3 -= cd1;        // 不可能的移除不算是错误
    }
}

```

如语句“`cd3 += cd1;`”所示, 委托可以多次出现在一个调用列表中。这种情况下, 它每出现一次, 就会被调用一次。在这样的调用列表中, 当移除委托时, 实际上移除的是调用列表中最后出现的那个委托实例。

就在执行最后一条语句“`cd3 -= cd1;`”之前, 委托 `cd3` 引用了一个空的调用列表。试图从空的列表中移除委托 (或者从非空列表中移除表中没有的委托) 不算是错误。

产生的输出为:

```

C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60

```

第 16 章 异常

C# 中的异常用于处理系统级和应用程序级的错误状态，它是一种结构化的、统一的和类型安全的处理机制。C# 中的异常机制非常类似于 C++ 的异常机制，但是有一些重要的区别：

- 在 C# 中，所有的异常必须由从 `System.Exception` 派生的类类型的实例来表示。在 C++ 中，可以使用任何类型的任何值表示异常。
- 在 C# 中，利用 `finally` 块 (§8.10) 可编写在正常执行和异常情况下都将执行的终止代码。在 C++ 中，很难在不重复代码的情况下编写这样的代码。
- 在 C# 中，系统级的异常如溢出、被零除和 `null` 等，都对应地定义了与其匹配的异常类，并且与应用程序级的错误状态处于同等地位。

16.1 导致异常的原因

可以以两种不同的方式引发异常。

- `throw` 语句 (§8.9.5) 用于立即无条件地引发异常。控制永远不会到达紧跟在 `throw` 后面的语句。
- 在处理 C# 语句和表达式的过程中，有时会出现一些例外情况，使某些操作无法正常完成，此时就会引发一个异常。例如，整数除法运算 (§7.7.2) 中，如果分母为零，就会引发一个 `System.DivideByZeroException`。有关可能以此方式引发的各种异常的列表，请参见 §16.4。

16.2 `System.Exception` 类

`System.Exception` 类是所有异常的基类型。此类具有一些所有异常共享的值得注意的属性：

- `Message` 是 `string` 类型的一个只读属性，它包含关于所发生异常的原因的描述（人工可读的^{注 1}）。
- `InnerException` 是 `Exception` 类型的一个只读属性。如果它的值不是 `null`，则它所引用的是导致了当前异常的那个异常，即表示当前异常是在处理那个 `InnerException` 的 `catch` 块中被引发的。否则，它的值为 `null`，则表示该异常不是由另一个异常引发的。以这种方式链接在一起的异常对象的数目可以是任意的。

^{注 1} 这里的英文为 “human-readable”，相对于 “machine-readable”，机器可读的。

这些属性的值可以在调用 `System.Exception` 的实例构造函数时指定。

16.3 异常的处理方式

异常是由 `try` 语句 (§8.10) 处理的。

发生异常时，系统将搜索可以处理该异常的最近的 `catch` 子句（根据该异常的运行时类型来选择）。首先，搜索当前的方法以查找一个词法上封闭着它的 `try` 语句，并按顺序考察与该 `try` 语句相关联的每个 `catch` 子句。如果上述操作失败，则在调用了当前方法的方法中，搜索在词法上封闭的着当前方法调用代码位置的 `try` 语句。此搜索将一直进行下去，直到找到可以处理当前异常的 `catch` 子句（该子句指定一个异常类，它与当前引发该异常的运行时类型属于同一个类或是该运行时类型所属类的一个基类）。注意，没有指定异常类的 `catch` 子句可以处理任何异常。

找到匹配的 `catch` 子句后，系统将把控制转移到该 `catch` 子句的第一条语句。在 `catch` 子句的执行开始前，系统将首先按顺序执行嵌套在捕捉到该异常的 `try` 语句里面的所有 `try` 语句所对应的全部 `finally` 子句。

如果没有找到匹配的 `catch` 子句，则发生下列两种情况之一：

- 如果对匹配的 `catch` 子句的搜索到达一个静态构造函数 (§10.11) 或静态字段初始值设定项，则在导致调用该静态构造函数的代码位置引发 `System.TypeInitializationException`。该 `System.TypeInitializationException` 的内部异常将包含最初引发的异常。
- 如果对匹配的 `catch` 子句的搜索到达最初启动当前线程的代码处，则该线程的执行就会终止。至于这样终止线程会造成什么影响要由具体的实现来确定。

特别值得注意的是在析构函数执行过程中发生的异常。如果在析构函数执行过程中引发异常且该异常未被捕获，则将终止该析构函数的执行，并调用它的基类的析构函数（如果有的话）。如果没有基类（如 `object` 类型中的情况），或者如果没有基类析构函数，则该异常将被忽略。

16.4 公共异常类

下列异常由某些 C# 操作引发。

表 16.1 公共异常类

异 常	说 明
<code>System.ArithmeticException</code>	在 算 术 运 算 期 间 发 生 的 异 常 （ 如 <code>System.DivideByZeroException</code> 和 <code>System.OverflowException</code> ）的基类
<code>System.ArrayTypeMismatchException</code>	当存储一个数组时，如果由于被存储的元素的实际类型与数组的实际类型不兼容而导致存储失败，就会抛出此异常
<code>System.DivideByZeroException</code>	在试图用零除整数值时抛出

(续表)

异 常	说 明
System.IndexOutOfRangeException	在试图使用小于零或超出数组界限的下标索引数组时抛出
System.InvalidCastException	当从基类型或接口到派生类型的显式转换在运行时失败时，就会抛出此异常
System.NullReferenceException	在需要使用引用对象的场合，如果使用 null 引用，就会抛出此异常
System.OutOfMemoryException	在分配内存（通过 new）的尝试失败时抛出
System.OverflowException	在 checked 上下文中的算术运算溢出时抛出
System.StackOverflowException	当执行堆栈由于保存了太多挂起的方法调用而耗尽时，就会抛出此异常。这通常表明存在非常深或无限的递归
System.TypeInitializationException	在静态构造函数引发异常并且没有可以捕捉到它的 catch 子句时

第 17 章 特性

C#语言的一个重要特征是使程序员能够为程序中定义的各种实体附加一些说明性信息。例如，类中方法的可访问性是通过用方法修饰符 `public`，`protected`，`internal` 和 `private` 来说明它而指定的。

C#使程序员可以创造说明性信息的新的种类，称为特性（`attributes`）。然后，程序员可以将这种特性附加到各种程序实体，而且在运行时环境中还可以检索这些特性信息。例如，一个框架可以定义一个名为 `HelpAttribute` 的特性，该特性可以放在某些程序元素（如类和方法）上，以提供从这些程序元素到其文档说明的映射。

特性是通过特性类（§17.1）的声明定义的，特性类可以具有定位和命名参数（§17.1.2）。特性是使用特性规范（§17.2）附加到 C#程序中的实体上的，而且可以在运行时作为特性实例（§17.3）检索。

17.1 特性类

从抽象类 `System.Attribute` 派生的类（不论是直接的还是间接的）称为特性类（`attribute class`）。关于特性类的声明定义一种新特性，它可以被放置在其他声明上。按照约定，特性类的名称带有 `Attribute` 后缀。使用特性时可以包含或省略此后缀。

17.1.1 特性用法

`AttributeUsage` 特性（§17.4.1）用于描述使用特性类的方式。

`AttributeUsage` 具有一个定位参数（§17.1.2），该参数使特性类能够指定自己可以用在那种声明上。示例：

```
using System;
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

定义了一个名为 `SimpleAttribute` 的特性类，此特性类只能放在类声明和接口声明上。示例：

```
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

显示了 `Simple` 特性的几种用法。虽然此特性是用名称 `SimpleAttribute` 定义的，但在使用时可以省略 `Attribute` 后缀，从而得到简称 `Simple`。因此，上例在语义上等效于：

```
[SimpleAttribute] class Class1 {...}
[SimpleAttribute] interface Interface1 {...}
```

AttributeUsage 还具有一个名为 **AllowMultiple** 的命名参数 (§17.1.2)，此参数用于说明对于某个给定实体，是否可以多次使用该特性。如果特性类的 **AllowMultiple** 为 **true**，则此特性类是多次性特性类，可以在一个实体上多次被应用；如果特性类的 **AllowMultiple** 为 **false** 或未指定的，则此特性类是一次性特性类，在一个实体上最多只能使用一次。

示例：

```
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;
    public AuthorAttribute(string name) {
        this.name = name;
    }
    public string Name {
        get { return name; }
    }
}
```

定义了一个名为 **AuthorAttribute** 的多次性特性类。示例：

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

显示了一个两次使用 **Author** 特性的类声明。

AttributeUsage 具有另一个名为 **Inherited** 的命名参数，此参数指示在基类上指定该特性时，该特性是否也会被从此基类派生的类所继承。如果特性类的 **Inherited** 为 **true**，则该特性会被继承。如果特性类的 **Inherited** 为 **false** 或者未指定，那么该特性不会被继承。

没有附加 **AttributeUsage** 特性的特性类 **X**，例如

```
using System;
class X: Attribute {...}
```

等效于下面的内容：

```
using System;
[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

17.1.2 定位和命名参数

特性类可以具有定位参数 (**positional parameter**) 和命名参数 (**named parameter**)。特性类的每个公共实例构造函数为该特性类定义一个有效的定位参数序列。特性类的每个非

静态公共读写字段和特性为该特性类定义一个命名参数。

示例：

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {        // 定位参数
        ...
    }
    public string Topic {                    // 命名参数
        get {...}
        set {...}
    }
    public string Url {
        get {...}
    }
}
```

定义了一个名为 `HelpAttribute` 的特性类，它具有一个定位参数 (`url`) 和一个命名参数 (`Topic`)。虽然 `Url` 特性是非静态的和公共的，但由于它不是读写的，因此它并不定义命名参数。

此特性类可以如下方式使用：

```
[Help("http://www.example.com/.../Class1.htm")]
class Class1
{
    ...
}
[Help("http://www.example.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

17.1.3 特性参数类型

特性类的定位参数和命名参数的类型仅限于特性参数类型 (attribute parameter type)，它们是：

- 下列类型之一：bool, byte, char, double, float, int, long, sbyte, short, string, uint, ulong, ushort。
- 类型 object。
- 类型 System.Type。
- 枚举类型，前提是该枚举类型具有 public 可访问性，而且所有嵌套着它的类型（如果有的话）也必须具有 public 可访问性 (§17.2)。
- 以上类型的一维数组。

17.2 特性专用化

特性专用化 (attribute specification) 就是将以前定义的特性应用到某个声明上。特性本身是一段附加说明性信息，可以把它指定给某个声明。可以在全局范围指定特性（即在包含程序集或模块上指定特性），也可以为下列各项指定特性：类型声明 (§9.5)、类成员说明 (§10.2)、接口成员声明 (§13.2)、结构成员声明 (§11.2)、枚举成员声明 (§14.3)、特性的访问器声明 (§10.6.2)、事件访问器声明 (§10.7.1) 和形参表 (§10.5.1)。

特性是在特性节中指定的。特性节由一对方括号组成，此方括号括着一个用逗号分隔的、含有一个或多个特性的列表。在这类列表中以何种顺序指定特性，以及附加到同一程序实体的特性节以何种顺序排列等细节并不重要。例如，特性专用化 [A][B], [B][A], [A, B] 和 [B, A] 是等效的。

global-attributes: (全局特性:)

global-attribute-sections (全局特性节)

global-attribute-sections: (全局特性节:)

global-attribute-section (全局特性节)

global-attribute-sections global-attribute-section (全局特性节 全局特性节)

global-attribute-section: (全局特性节:)

[global-attribute-target-specifier attribute-list] ([全局特性目标说明符 特性列表])

[global-attribute-target-specifier attribute-list ,] ([全局特性目标说明符 特性列表 ,])

global-attribute-target-specifier: (全局特性目标说明符:)

global-attribute-target : (全局特性目标 :)

global-attribute-target: (全局特性目标:)

assembly

module

attributes: (特性:)

attribute-sections (特性节)

attribute-sections: (特性节:)

attribute-section (特性节)

attribute-sections attribute-section (特性节 特性节)

attribute-section: (特性节:)

[attribute-target-specifier_{opt} attribute-list] ([特性目标说明符_{可选} 特性列表])

[attribute-target-specifier_{opt} attribute-list ,] ([特性目标说明符_{可选} 特性列表 ,])

attribute-target-specifier: (特性目标说明符:)

attribute-target : (特性目标 :)

attribute-target: (特性目标:)

```

field
event
method
param
property
return
type

```

attribute-list: (特性列表:)

attribute (特性)

attribute-list , **attribute** (特性列表 , 特性)

attribute: (特性:)

attribute-name **attribute-arguments**_{opt} (特性名 特性参数_{可选})

attribute-name: (特性名:)

type-name (类型名)

attribute-arguments: (特性参数:)

(**positional-argument-list**_{opt}) ((定位参数列表_{可选}))

(**positional-argument-list** , **named-argument-list**) ((定位参数列表 , 命名参数列表))

(**named-argument-list**) ((命名参数列表))

positional-argument-list: (定位参数列表:)

positional-argument (定位参数)

positional-argument-list , **positional-argument** (定位参数列表 , 定位参数)

positional-argument: (定位参数:)

attribute-argument-expression (特性参数表达式)

named-argument-list: (命名参数列表:)

named-argument (命名参数)

named-argument-list , **named-argument** (命名参数列表 , 命名参数)

named-argument: (命名参数:)

identifier = **attribute-argument-expression** (标识符 = 特性参数表达式)

attribute-argument-expression: (特性参数表达式:)

expression (表达式)

如上所述, 特性由一个特性名和一个可选的定位和命名参数列表组成。定位参数(如果有的话)列在命名参数前面。定位参数包含一个特性参数表达式; 命名参数包含一个名称, 名称后跟一个等号和一个特性参数表达式。这两种参数都受简单赋值规则约束。命名参数的排列顺序无关紧要。

特性名用于标识特性类。如果特性名的形式等同于一个类型名, 则此名称必须引用一个特性类。否则将发生编译时错误。示例:

```

class Class1 {}
[Class1] class Class2 {} // 错误

```

产生编译时错误, 因为它试图将 Class1 用做特性类, 而 Class1 并不是一个特性类。

某些上下文允许将一个特性指定给多个目标。程序中可以利用“特性目标说明符”来显式地指定目标。特性放置在全局级别中时，则需要全局特性目标说明符。对于所有其他位置上的特性，则采用系统提供的合理的默认值，但是在某些目标不明确的情况下可以使用特性目标说明符来确认或重写默认值，也可以在目标明确的情况下使用特性目标说明符来确认默认值。因此，除非是在全局级别中，否则通常可以省略特性目标说明符。对于可能造成不明确性的上下文，按下述规则处理：

- 在全局范围指定的特性可以应用于目标程序集或目标模块。系统没有为此上下文提供默认形式，所以在此上下文中始终需要一个特性目标说明符。如果存在 `assembly` 特性目标说明符，则表明此特性适用于指定的目标程序集；如果存在 `module` 特性目标说明符，则表明此特性适用于指定的目标模块。
- 在委托声明上指定的特性，或者适用于所声明的委托，或者适用于它的返回值。如果不存在特性目标说明符，则此特性适用于该委托。如果存在 `type` 特性目标说明符，则表明此特性适用于该委托；如果存在 `return` 特性目标说明符，则表明此特性适用于该返回值。
- 在方法声明上指定的特性，或者适用于所声明的方法，或者适用于它的返回值。如果不存在特性目标说明符，则此特性适用于该方法。如果存在 `method` 特性目标说明符，则表明此特性适用于该方法；如果存在 `return` 特性目标说明符，则表明此特性适用于该返回值。
- 在运算符声明上指定的特性，或者适用于所声明的运算符，或者适用于它的返回值。如果不存在特性目标说明符，则此特性适用于该运算符。如果存在 `method` 特性目标说明符，则表明此特性适用于该运算符；如果存在 `return` 特性目标说明符，则表明此特性适用于该返回值。
- 对于在省略了事件访问器的事件声明上指定的特性，它的目标对象有三种可能的选择：所声明的事件；与该事件关联的字段（如果该事件是非抽象事件的话）；与该事件关联的 `add` 和 `remove` 方法。如果不存在特性目标说明符，则此特性适用于该事件。如果存在 `event` 特性目标说明符，则表明此特性适用于该事件；如果存在 `field` 特性目标说明符，则表明此特性适用于该字段；而如果存在 `method` 特性目标说明符，则表明此特性适用于这些方法。
- 在特性或索引器声明中的 `get` 访问器声明上指定的特性，或者适用于该访问器关联的方法，或者适用于它的返回值。如果不存在特性目标说明符，则此特性适用于该方法。如果存在 `method` 特性目标说明符，则表明此特性适用于该方法；如果存在 `return` 特性目标说明符，则表明此特性适用于该返回值。
- 在属性或索引器声明中的 `set` 访问器上指定的特性，或者可适用于该访问器关联的方法，或者适用于它的独立的隐式参数。如果不存在特性目标说明符，则此特性适用于该方法。如果存在 `method` 特性目标说明符，则此特性适用于该方法；如果存在 `param` 特性目标说明符，则此特性适用于该参数。
- 在事件声明的添加或移除访问器声明上指定的特性，或者适用于该访问器关联的方法，或者适用于它的独立参数。如果不存在特性目标说明符，则此特性适用于该方法。如果存在 `method` 特性目标说明符，则此特性适用于该方法；如果存在

`param` 特性目标说明符，则此特性适用于该参数。

在其他上下文中，允许包含一个特性目标说明符，但这样做是没有必要的。例如，类声明中的特性既可以包括也可以省略说明符 `type`：

```
[type: Author("Brian Kernighan")]
class Class1 {}
[Author("Dennis Ritchie")]
class Class2 {}
```

如果指定了无效的特性目标说明符，则会发生错误。例如，不能将说明符 `param` 用在类声明中：

```
[param: Author("Brian Kernighan")] // 错误
class Class1 {}
```

按照约定，特性类的名称均带有 `Attribute` 后缀。类型名形式的特性名既可以包含也可以省略此后缀。如果发现特性类中同时出现带和不带此后缀的名称，则引用时就可能出现多义性，从而导致运行时错误。如果在拼写特性名时，明确说明其最右边的标识符为逐字标识符 (§2.4.2)，则它仅匹配没有后缀的特性，从而能够解析这类多义性。示例：

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}
[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}
[X] // 错误：多义性
class Class1 {}
[XAttribute] // 引用 XAttribute
class Class2 {}
[@X] // 引用 X
class Class3 {}
[@XAttribute] // 引用 XAttribute
class Class4 {}
```

显示两个名字分别为 `X` 和 `XAttribute` 的特性类。特性 `[X]` 含义不明确，因为它既可引用 `X` 也可引用 `XAttribute`。使用逐字标识符能够在这种极少见的情况下表明确切的意图。特性 `[XAttribute]` 是明确的（尽管在存在名为 `XAttributeAttribute` 的特性类时它将是明确的！）。如果移除了类 `X` 的声明，那么上述两个特性都将引用名为 `XAttribute` 的特性类，如下所示：

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}
[X] // 引用 XAttribute
class Class1 {}
[XAttribute] // 引用 XAttribute
class Class2 {}
[@X] // 错误，没有名为 X 的特性
class Class3 {}
```

在同一个实体中多次使用一次性特性类会导致编译时错误。示例：

```

using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;
    public HelpStringAttribute(string value) {
        this.value = value;
    }
    public string Value {
        get {...}
    }
}
[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}

```

导致编译时错误，因为它试图在 Class1 的声明中多次使用一次性特性类 HelpString。

如果表达式 E 符合下列所有条件的话，则表达式 E 为“特性参数表达式”：

- E 的类型是特性参数类型 (§17.1.3)。
- 在编译时，E 的值可以解析为下列之一：
 - 常数值。
 - System.Type 对象。
 - 特性参数表达式的一维数组。

例如：

```

using System;
[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }
    public Type P2 {
        get {...}
        set {...}
    }
    public object P3 {
        get {...}
        set {...}
    }
}
[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}

```

17.3 特性实例

特性实例 (attribute instance) 是一种用于在运行时表示特性的实例。特性是用特性类、定位参数和命名参数定义的。特性实例是该特性类的一个实例，它是用定位参数和命名参数初始化后得到的。

特性实例的检索涉及编译时和运行时处理，详见后面几节中的介绍。

17.3.1 特性的编译

对于一个具有特性类 `T`、定位参数列表 `P` 和命名参数列表 `N` 的特性的编译过程由下列步骤组成。

- 遵循形式为 `new T(P)` 的对象创建表达式编译规则所规定的步骤进行编译时处理。这些步骤或者导致编译时错误，或者确定 `T` 上的可以在运行时调用的实例构造函数 `C`。
- 如果 `C` 不具有公共可访问性，则发生编译时错误。
- 对于 `N` 中的每个命名参数 `Arg`：
 - 将 `Name` 设为命名参数 `Arg` 的标识符。
 - `Name` 必须标识 `T` 中的一个非静态读写 `public` 字段或特性。如果 `T` 没有这样的字段或特性，则发生编译时错误。
- 保存下列信息：特性类 `T`、`T` 上的实例构造函数 `C`、定位参数列表 `P` 和命名参数列表 `N`，以供在运行时实例化该特性时调用。

17.3.2 特性实例的运行时检索

对一个特性进行编译后，会产生一个特性类 `T`、一个 `T` 上的实例构造函数 `C`、一个定位参数列表 `P` 和一个命名参数列表 `N`。给定了上述信息后，就可以在运行时使用下列步骤进行检索来生成一个特性实例。

- 遵循执行 `new T(P)` 形式的对象创建表达式（使用在编译时确定的实例构造函数 `C`）的运行时处理步骤。这些步骤或者导致异常，或者产生 `T` 的一个实例 `O`。
- 对于 `N` 中的每个命名参数 `Arg`，按以下顺序进行处理：
 - 将 `Name` 设为命名参数 `Arg` 的标识符。如果 `Name` 标识的不是 `O` 上的一个非静态读写 `public` 字段或特性，则引发异常。
 - 将 `Value` 设为 `Arg` 的特性参数表达式的计算结果。
 - 如果 `Name` 标识 `O` 上的一个字段，则将此字段赋值为 `Value`。
 - 否则，`Name` 就标识 `O` 上的一个特性。将此特性设置为 `Value`。
 - 结果为 `O`，它是已经用定位参数列表 `P` 和命名参数列表 `N` 初始化了的特性类 `T` 的一个实例。

17.4 保留特性

少数特性以某种方式影响语言。这些特性包括：

- `System.AttributeUsageAttribute` (§17.4.1)，它用于描述可以以何种方式使用特性类。
- `System.Diagnostics.ConditionalAttribute` (§17.4.2)，它用于定义条件方法。
- `System.ObsoleteAttribute` (§17.4.3)，它用于将某个成员标记为已过时。

17.4.1 AttributeUsage 特性

AttributeUsage 特性用于描述使用特性类的方式。

用 AttributeUsage 特性修饰的类必须直接或间接从 System.Attribute 派生。否则将发生编译时错误。

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute: Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
        public virtual AttributeTargets ValidOn { get {...} }
    }
    public enum AttributeTargets
    {
        Assembly      = 0x0001,
        Module        = 0x0002,
        Class          = 0x0004,
        Struct         = 0x0008,
        Enum           = 0x0010,
        Constructor    = 0x0020,
        Method         = 0x0040,
        Property       = 0x0080,
        Field          = 0x0100,
        Event          = 0x0200,
        Interface      = 0x0400,
        Parameter      = 0x0800,
        Delegate       = 0x1000,
        ReturnValue     = 0x2000,
        All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate | ReturnValue
    }
}
```

17.4.2 Conditional 特性

利用 Conditional 特性,程序员可以定义条件方法(conditional method)。Conditional 特性通过测试条件编译符号来确定适用的条件。当运行到一个条件方法调用时,是否执行该调用,要根据出现该调用时是否已定义了此符号来确定。如果定义了此符号,则执行该调用;否则省略该调用(包括对调用的参数的计算)。

```
namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
    }
}
```

条件方法要受到以下限制:

- 条件方法必须是类声明或结构声明中的方法。如果在接口声明中的方法上指定 `Conditional` 特性, 将出现编译时错误。
- 条件方法必须具有 `void` 返回类型。
- 不能用 `override` 修饰符标识条件方法。但是, 可以用 `virtual` 修饰符标识条件方法。此类方法的重写方法隐含为有条件的方法, 而且不能用 `Conditional` 特性显式标识。
- 条件方法不能是接口方法的实现。否则将发生编译时错误。

此外, 如果条件方法用在委托创建表达式中, 也会发生编译时错误。示例:

```
#define DEBUG
using System;
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}
class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

将 `Class1.M` 声明为条件方法。`Class2` 的 `Test` 方法调用此方法。由于定义了条件编译符号 `DEBUG`, 因此如果调用 `Class2.Test`, 那么它将调用 `M`。如果尚未定义符号 `DEBUG`, 那么 `Class2.Test` 将不会调用 `Class1.M`。

一定要注意包含或排除对条件方法的调用是由该调用所在处的条件编译符号控制的。

文件 `class1.cs`:

```
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

文件 `class2.cs`:

```
#define DEBUG
class Class2
{
    public static void G() {
        Class1.F(); // F 被调用
    }
}
```

文件 `class3.cs`:

```
#undef DEBUG
class Class3
{
    public static void H() {
        Class1.F();          // F 不被调用
    }
}
```

其中，类 `Class2` 和 `Class3` 分别包含对条件方法 `Class1.F` 的调用，根据是否定义了 `DEBUG`，此调用是有条件的。由于在 `Class2` 而不是 `Class3` 的上下文中定义了此符号，因此在此在 `Class2` 中包含了对 `F` 的调用，而在 `Class3` 中省略了对 `F` 的调用。

在继承链中使用条件方法可能引起混乱。通过 `base.M` 形式的 `base` 对条件方法进行的调用受正常条件方法调用规则的限制。

文件 `class1.cs`:

```
using System;
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

文件 `class2.cs`:

```
using System;
class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M();          // base.M 不被调用!
    }
}
```

文件 `class3.cs`:

```
#define DEBUG
using System;
class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();              // M 被调用
    }
}
```

其中，`Class2` 包括一个对在其基类中定义的 `M` 的调用。此调用被省略，因为基方法是条件性的，依赖于符号 `DEBUG` 是否存在，而该符号在此处没有定义。因此，该方法仅向控制台写入“`Class2.M executed`”。审慎地使用 `pp` 声明（`pp-declarations`）可以消除这类问题。

17.4.3 Obsolete 特性

Obsolete 特性用于标记不应该再使用的类型和类型成员。

```
namespace System
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute: Attribute
    {
        public ObsoleteAttribute() {...}
        public ObsoleteAttribute(string message) {...}
        public ObsoleteAttribute(string message, bool error) {...}
        public string Message { get {...} }
        public bool IsError { get {...} }
    }
}
```

如果程序使用由 **Obsolete** 特性修饰的类型或成员，则编译器将发出警告或错误信息。具体说来，如果没有提供错误参数，或者如果提供了错误参数但该错误参数的值为 **false**，则编译器将发出警告。如果指定了错误参数并且该错误参数的值为 **true**，则会引发一个编译时错误。

对于下面的示例：

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}
class B
{
    public void F() {}
}
class Test
{
    static void Main() {
        A a = new A();    // 警告
        a.F();
    }
}
```

其中，类 **A** 是用 **Obsolete** 特性修饰的。在 **Main** 的代码中，每次使用 **A** 时均会导致一个包含指定信息 “This class is obsolete; use class B instead” 的警告。

17.5 互操作的特性

17.5.1 与 COM 和 Win32 组件的互操作

.NET 运行库提供了大量特性，这些特性使 C# 程序能够与使用 COM 和 Win32 DLL 编写的组件交互操作。例如，可以在 `static extern` 方法上使用 `DllImport` 特性来表示该方法的实现应该到 Win32 DLL 中去查找。这些特性可在 `System.Runtime.InteropServices` 命名空间中找到，在 .NET 运行库文档中可以找到关于这些特性的详细文档。

17.5.2 与其他 .NET 语言的交互操作

索引器是利用索引特性在 .NET 中实现的，并且具有一个属于 .NET 元数据的名称。如果索引器没有被指定 `IndexerName` 特性，则默认情况下将使用名称 `Item`。`IndexerName` 特性使开发人员可以重写此默认名称并指定不同的名称。

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

第 18 章 不安全代码

如前面几章所定义，核心 C# 语言没有将指针列入它所支持的数据类型，从而与 C 和 C++ 有着显著的区别。作为替代，C# 提供了各种引用类型，并能够创建可由垃圾回收器管理的对象。这种设计结合其他功能，使 C# 成为比 C 或 C++ 安全得多的语言。在核心 C# 语言中，干脆不可能有未初始化的变量、“虚”指针或者超过数组的边界对其进行索引的表达式。这样，以往总是不断地烦扰 C 和 C++ 程序的一系列错误就不会再出现了。

尽管实际上对 C 或 C++ 中的每种指针类型构造，C# 都设置了与之对应的引用类型，但仍然会有一些场合需要访问指针类型。例如，当需要与底层操作系统进行交互、访问内存映射设备，或实现一些以时间为关键的算法时，若没有访问指针的手段，就不可能或者至少很难完成。为了满足这样的需求，C# 提供了编写不安全代码的能力。

在不安全代码中，可以声明和操作指针，可以在指针和整型之间执行转换，还可以获取变量的地址，等等。在某种意义上，编写不安全代码很像在 C# 程序中编写 C 代码。

无论从开发人员还是从用户角度来看，不安全代码事实上都是一种“安全”功能。不安全代码必须用修饰符 `unsafe` 明确地标记，这样开发人员就不会误用不安全功能，而执行引擎将确保不会在不受信任的环境中执行不安全代码。

18.1 不安全上下文

C# 的不安全功能仅用于不安全上下文中。不安全上下文是通过在类型或成员的声明中包含一个 `unsafe` 修饰符或者通过使用不安全语句引入的：

- 类、结构、接口或委托的声明可以包含 `unsafe` 修饰符，在这种情况下，该类型声明的整个文本范围（包括类、结构或接口的体）被认为是不安全上下文。
- 在字段、方法、属性、事件、索引器、运算符、实例构造函数、析构函数或静态构造函数的声明中，也可以包含 `unsafe` 修饰符，在这种情况下，该成员声明的整个文本范围被认为是不安全上下文。
- 不安全语句使得可以在块内使用不安全上下文。该语句关联的块的整个文本范围被认为是不安全上下文。

下面显示了关联的语法扩展。为简洁起见，用省略号 (...) 表示前几章中出现过的产生式。

`class-modifier`: (类修饰符:)

...

`unsafe`

`struct-modifier`: (结构修饰符:)

```
...
unsafe
interface-modifier: (接口修饰符:)
...
unsafe
delegate-modifier: (委托修饰符:)
...
unsafe
field-modifier: (字段修饰符:)
...
unsafe
method-modifier: (方法修饰符:)
...
unsafe
property-modifier: (属性修饰符:)
...
unsafe
event-modifier: (事件修饰符:)
...
unsafe
indexer-modifier: (索引器修饰符:)
...
unsafe
operator-modifier: (运算符修饰符:)
...
unsafe
constructor-modifier: (构造函数修饰符:)
...
unsafe
destructor-declaration: (析构函数声明:)
attributesopt externopt unsafeopt ~ identifier ( ) destructor-body (特性opt
选 extern 可选 unsafe 可选 ~ 标识符 ( ) 析构函数体)
attributesopt unsafeopt externopt ~ identifier ( ) destructor-body (特性opt
选 unsafe 可选 extern 可选 ~ 标识符 ( ) 析构函数体)
static-constructor-modifier: (静态构造函数修饰符:)
externopt      unsafeopt  static
unsafeopt      externopt  static
externopt      static    unsafeopt
unsafeopt      static    externopt
```

```

        static      externopt   unsafeopt
        static      unsafeopt   externopt

```

embedded-statement: (嵌入语句:)

...

unsafe-statement (不安全语句)

unsafe-statement: (不安全语句:)

unsafe block (unsafe 块)

对于下面的示例:

```

public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}

```

在结构声明中指定的 **unsafe** 修饰符导致该结构声明的整个文本范围成为不安全上下文。因此, 可以将 **Left** 和 **Right** 字段声明为指针类型的。上面的示例还可以编写为:

```

public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}

```

此处, 字段声明中的 **unsafe** 修饰符导致这些声明被认为是不安全上下文。

除了建立不安全上下文从而允许使用指针类型外, **unsafe** 修饰符对类型或成员没有影响。下面的示例中:

```

public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}
public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}

```

A 中 F 方法上的 **unsafe** 修饰符直接导致 F 的文本范围成为不安全上下文并可以在其中使用语言的不安全功能。在 B 中对 F 的重写中, 不需要重新指定 **unsafe** 修饰符, 除非 B 中的 F 方法本身需要访问不安全功能。

当指针类型是方法签名的一部分时, 情况略有不同:

```

public unsafe class A
{
    public virtual void F(char* p) {...}
}

```

```
public class B: A
{
    public unsafe override void F(char* p) {...}
}
```

此处，由于 F 的签名包括指针类型，因此它只能出现在不安全上下文中。然而，为设置此不安全上下文，既可以将整个类设置为不安全的（如 A 中的情况），也可以仅在方法声明中包含一个 unsafe 修饰符（如 B 中的情况）。

18.2 指针类型

在不安全上下文中，类型 (§4) 可以是指针类型，或者是值类型或引用类型。

type: (类型:)

- value-type (值类型)
- reference-type (引用类型)
- pointer-type (指针类型)

指针类型可表示为非托管类型后跟一个 “*” 标记，或者关键字 void 后跟一个 “*” 标记:

- pointer-type: (指针类型:)
 - unmanaged-type * (非托管类型 *)
 - void * (void *)
- unmanaged-type: (非托管类型)
 - type (类型)

指针类型中，在 “*” 前面指定的类型称为该指针类型的目标类型 (referent type)。它表示该指针类型的值所指向的变量的类型。

与引用 (引用类型的值) 不同，指针不受垃圾回收器跟踪 (垃圾回收器不了解指针和它们指向的数据)。出于此原因，不允许指针指向引用或者包含引用的结构，并且指针的目标类型必须是非托管类型。

非托管类型是任何不是引用类型，并且在任何嵌套级别都不包含“引用类型”字段的类型。换句话说，非托管类型是下列类型之一:

- sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal 或 bool。
- 任何枚举类型。
- 任何指针类型。
- 任何由用户定义的只包含非托管类型字段的结构类型。

将指针和引用进行混合使用时的基本规则是: 引用 (对象) 的目标可以包含指针，但指针的目标不能包含引用。

表 18.1 给出了一些指针类型的示例:

表 18.1 指针类型示例

示 例	说 明
byte*	指向 byte 的指针
char*	指向 char 的指针
int**	指向 int 的指针的指针
int*[]	一维数组，它的元素是指向 int 的指针
void*	指向未知类型的指针

对于某个给定实现，所有的指针类型都必须具有相同的大小和表示形式。

与 C 和 C++ 不同，在 C# 中，当在同一声明中声明多个指针时，“*” 只与基础类型写在一起，而不充当每个指针名称的前缀标点符号。例如：

```
int* pi, pj; // 与 int *pi, *pj 不同;
```

类型为 T* 的一个指针的值表示类型为 T 的一个变量的地址。指针间接寻址运算符 * (§18.5.1) 可用于访问此变量。例如，假设具有 int* 类型的变量 P，则表达式 *P 表示一个 int 变量，该变量的地址就是 P 的值。

与对象引用类似，指针可以是 null。如果将间接寻址运算符应用于 null 指针，则其行为将由实现自己定义。值为 null 的指针表示将该指针的所有位都置零。

void* 类型表示指向未知类型的指针。由于目标类型是未知的，因此间接寻址运算符不能应用于 void* 类型的指针，也不能对这样的指针执行任何算术运算。但是 void* 类型的指针可以强制转换为任何其他指针类型（反之亦然）。

指针类型是一个单独类别的类型。与引用类型和值类型不同，指针类型不从 object 继承，而且不存在指针类型和 object 之间的转换。具体说来，指针不支持装箱和取消装箱 (§4.3) 操作。但是，允许在不同指针类型之间及指针类型与整型之间进行转换。§18.4 对此进行了描述。

指针类型可用做易失字段的类型 (§10.4.3)。

虽然指针可以作为 ref 或 out 参数传递，但这样做可能会导致未定义的行为，例如，指针可能被设置为指向一个局部变量，而当调用方法返回时，该局部变量可能已不存在了；或者指针曾指向一个固定对象，但当调用方法返回时，该对象不再是固定的了。例如：

```
using System;
class Test
{
    static int value = 20;
    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;
        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }
    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
```

```

    int* px2 = &i;
    F(out px1, ref px2);
    Console.WriteLine("*px1 = {0}, *px2 = {1}",
        *px1, *px2); // 不明确的行为
}
}

```

方法可以返回某一类型的值，而该类型可以是指针。例如，给定一个指向连续的 `int` 值序列的指针、该序列的元素个数，和另外一个 `int` 值，下面的方法将在该整数序列中查找与该值匹配的值，若找到匹配项，则返回该匹配项的地址；否则，它将返回 `null`：

```

unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}

```

在不安全上下文中，可以使用下列几种构造操作指针：

- `*` 运算符可用于执行指针间接寻址 (§18.5.1)。
- `->` 运算符可用于通过指针访问结构的成员 (§18.5.2)。
- `[]` 运算符可用于索引指针 (§18.5.3)。
- `&` 运算符可用于获取变量的地址 (§18.5.4)。
- `++` 运算符 和 `--` 运算符可用于增加和减小指针 (§18.5.5)。
- `+` 运算符 和 `-` 运算符可用于执行指针算术 (§18.5.6)。
- `==`, `!=`, `<`, `>`, `<=` 和 `=>` 运算符可用于比较指针 (§18.5.7)。
- `stackalloc` 运算符可用于从调用堆栈 (§18.7) 中分配内存。
- `fixed` 语句可用于暂时固定一个变量，以便可以获取它的地址 (§18.6)。

18.3 固定变量和可移动变量

`address-of` 运算符 (§18.5.4) 和 `fixed` 语句 (§18.6) 将变量划分为两个类别：固定变量 (`fixed variable`) 和可移动变量 (`moveable variable`)。

固定变量驻留在不受垃圾回收器的操作影响的存储位置中（固定变量的示例包括局部变量、值参数和由取消指针引用而创建的变量）。另一方面，可移动变量则驻留在会被垃圾回收器重定位或处置的存储位置中（可移动变量的示例包括对象中的字段和数组的元素）。

`&` 运算符 (§18.5.4) 允许不受限制地获取固定变量的地址。但是，由于可移动变量会受到垃圾回收器的重定位或处置，因此可移动变量的地址只能使用 `fixed` 语句 (§18.6) 获取，而且该地址只在此 `fixed` 语句的生存期内有效。

准确地说，固定变量是下列之一：

- 用引用局部变量或值参数的“简单名称” (§7.5.2) 表示的变量。
- 用 `V.I` 形式的“成员访问” (§7.5.4) 表示的变量，其中 `V` 是“结构类型”的固

定变量。

- 用 `*P` 形式的“指针间接寻址表达式” (§18.5.1)、`P->I` 形式的“指针成员访问” (§18.5.2) 或 `P[E]` 形式的“指针元素访问” (§18.5.3) 表示的变量。

所有其他变量都属于可移动变量。

请注意静态字段属于可移动变量。还请注意即使赋予 `ref` 或 `out` 参数的自变量是固定变量，它们仍属于可移动变量。最后请注意，由取消指针引用而产生的变量总是属于固定变量。

18.4 指针转换

在不安全上下文中，可供使用的隐式转换的集合 (§6.1) 也扩展为包括以下隐式指针转换：

- 从任何指针类型到 `void*` 类型。
- 从 `null` 类型到任何指针类型。

另外，在不安全上下文中，可供使用的显式转换的集合 (§6.2) 也扩展为包括以下显式指针转换：

- 从任何指针类型到任何其他指针类型。
- 从 `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` 或 `ulong` 到任何指针类型。
- 从任何指针类型到 `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` 或 `ulong`。

最后，在不安全上下文中，标准隐式转换的集合 (§6.3.1) 包括以下指针转换：

- 从任何指针类型到 `void*` 类型。

两个指针类型之间的转换永远不会更改实际的指针值。换句话说，从一个指针类型到另一个指针类型的转换不会影响由指针给出的基础地址。

当一个指针类型被转换为另一个指针类型时，如果没有将得到的指针正确地对指向的类型对齐，则当结果被取消引用时，该行为将是未定义的。一般情况下，“正确对齐”的概念是可传递的：如果指向类型 `A` 的指针被正确地与指向类型 `B` 的指针对齐，而此指向类型 `B` 的指针又被正确地与指向类型 `C` 的指针对齐，那么指向类型 `A` 的指针被正确地与指向类型 `C` 的指针对齐。

请考虑下列情况，其中具有一个类型的变量经由指向一个不同类型的指针访问：

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;      // 不明确
*pi = 123456;    // 不明确
```

当一个指针类型被转换为指向字节的指针时，转换后的指针将指向原来所指变量的地址中的最低寻址字节。连续增加该变换后的指针（最大可达到该变量所占内存空间的大小），将产生指向该变量的其他字节的指针。例如，下列方法将 `double` 型变量中的 8 个字节的每一个显示为一个十六进制值：

```

using System;
class Test
{
    static void Main() {
        double d = 123.456e23;
        unsafe {
            byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
                Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}

```

当然，产生的输出取决于字节存储顺序 (Endianness)。

指针和整数之间的映射由实现来定义。但是，在具有线性地址空间的 32 位和 64 位 CPU 体系结构上，指针和整型之间的转换通常与 uint 或 ulong 类型的值和这些整型之间的对应方向上的转换具有完全相同的行为。

18.5 表达式中的指针

在不安全上下文中，表达式可能产生指针类型的结果，但是在不安全上下文以外，表达式为指针类型会导致编译时错误。准确地说，在不安全上下文以外，如果任何简单名称 (§7.5.2)、成员访问 (§7.5.4)、调用表达式 (§7.5.5) 或元素访问 (§7.5.6) 属于指针类型，则将发生编译时错误。

在不安全上下文中，非数组创建基本表达式 (§7.5) 和一元表达式 (§7.6) 产生式允许使用下列附加构造：

primary-no-array-creation-expression: (非数组创建基本表达式:)

...

pointer-member-access (指针成员访问)

pointer-element-access (指针元素访问)

sizeof-expression (sizeof 表达式)

unary-expression: (一元表达式:)

...

pointer-indirection-expression (指针间接寻址表达式)

addressof-expression (addressof 表达式)

以下几节对这些构造进行了描述。相关的语法暗示了不安全运算符的优先级和结合性。

18.5.1 指针间接寻址

指针间接寻址表达式包含一个星号 (*), 后跟一个一元表达式。

pointer-indirection-expression: (指针间接寻址表达式:)

* unary-expression (* 一元表达式)

一元 * 运算符表示指针间接寻址并且用于获取指针所指向的变量。计算 *P 得到的结果（其中 P 为指针类型 T* 的表达式）是类型为 T 的一个变量。将一元 * 运算符应用于 void* 类型的表达式或者应用于不是指针类型的表达式会导致编译时错误。

将一元 * 运算符应用于 null 指针的效果是由实现定义的。具体说来，不能保证此操作会引发 System.NullReferenceException。

如果已经将无效值赋给指针，则一元 * 运算符的行为是未定义的。通过一元 * 运算符取消指针引用有时会产生无效值，这些无效值包括：没能按所指向的类型正确对齐的地址（请参见§18.4 中的示例）和超过生存期的变量的地址。

出于明确赋值分析的目的，通过计算 *P 形式的表达式产生的变量被认为是初始化赋过值的 (§5.3.1)。

18.5.2 指针成员访问

指针成员访问包含一个基本表达式，后跟一个“->”标记，最后是一个标识符。

pointer-member-access: (指针成员访问:)

primary-expression -> identifier (基本表达式 -> 标识符)

在 P->I 形式的指针成员访问中，P 必须是除 void* 以外的某个指针类型的表达式，而 I 必须表示 P 所指向的类型的可访问成员。

P->I 形式的指针成员访问的计算方式与 (*P).I 完全相同。有关指针间接寻址运算符 (*) 的说明，请参见§18.5.1。有关成员访问运算符 (.) 的说明，请参见§7.5.4。

下面的示例中：

```
using System;
struct Point
{
    public int x;
    public int y;
    public override string ToString() {
        return "(" + x + ", " + y + ")";
    }
}
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

-> 运算符用于通过指针访问结构中的字段和调用结构中的方法。由于 P->I 操作完全等效于 (*P).I，因此 Main 方法可以等效地编写为：

```
class Test
{

```

```

static void Main() {
    Point point;
    unsafe {
        Point* p = &point;
        (*p).x = 10;
        (*p).y = 20;
        Console.WriteLine((*p).ToString());
    }
}

```

18.5.3 指针元素访问

指针元素访问包括一个非数组创建基本表达式，后跟一个用“[]”括起来的表达式。

pointer-element-access: (指针元素访问:)

primary-no-array-creation-expression [expression] (非数组创建基本表达式 [表达式])

在 P[E] 形式的指针元素访问中，P 必须是 void* 以外的指针类型的表达式，而 E 则必须是可以隐式转换为 int, uint, long 或 ulong 的类型的表达式。

P[E] 形式的指针元素访问的计算方式与 *(P + E) 完全相同。有关指针间接寻址运算符 (*) 的说明，请参见§18.5.1。有关指针加法运算符 (+) 的说明，请参见§18.5.6。

下面的示例：

```

class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}

```

其中，指针元素访问用于在 for 循环中初始化字符缓冲区。由于 P[E] 操作与 *(P + E) 完全等效，因此此示例可以等效地写为：

```

class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}

```

指针元素访问运算符不能检验是否发生访问越界错误，而且当访问超出边界的元素时行为是未定义的。这与 C 和 C++ 相同。

18.5.4 address-of 运算符

address-of 表达式包含一个“与”符号 (&), 后跟一个一元表达式。

addressof-expression: (addressof 表达式:)

& unary-expression (& 一元表达式)

如果给定类型为 T 且属于固定变量 (§18.3) 的表达式 E , 构造 $\&E$ 将计算由 E 给出的变量的地址。计算的结果是一个类型为 T^* 的值。如果 E 不属于变量, 如果 E 属于易失字段, 或者如果 E 表示可移动的变量, 则发生编译时错误。在最后一种情况下, 可以先利用固定语句 (§18.6) 临时“固定”该变量, 再获取它的地址。

& 运算符不要求它的参数先被明确赋值, 但是在执行了 & 操作后, 该运算符所应用于的那个变量在此操作发生的执行路径中被认为是已经明确赋值的。这意味着, 由程序员负责确保在相关的上下文中对该变量实际进行适当的初始化。

对于下面的示例:

```
using System;
class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

初始化 p 的代码执行了 $\&i$ 操作, 此后 i 被认为是明确赋值的。对 $*p$ 的赋值实际上是初始化了 i , 但设置此初始化是程序员的责任, 而且如果移除此赋值语句, 不会发生编译时错误。

上述 & 运算符的明确赋值规则可以避免局部变量的冗余初始化。例如, 许多外部 API 要求获取指向结构的指针, 而由此 API 来填充该结构。对此类 API 进行的调用通常会传递局部结构变量的地址, 而如果没有上述规则, 则需要对此结构变量进行冗余初始化。

如 §7.5.4 中所述, 如果在实例构造函数或静态构造函数之外, 在结构或类中定义了 readonly 字段, 则该字段被认为是一个值, 而不是变量。因此, 无法获取它的地址。与此类似, 无法获取常数的地址。

18.5.5 指针增加和指针减小

在不安全上下文中, ++ 运算符 (§7.5.9) 和 -- 运算符 (§7.6.5) 可以应用于除 void^* 以外的所有类型的指针变量。因此, 为每个指针类型 T^* 都隐式定义了下列运算符:

```
T* operator ++(T* x);
T* operator --(T* x);
```

这些运算符分别产生与 $x + 1$ 和 $x - 1$ (§18.5.6) 相同的结果。换句话说, 对于 T^* 类型的指针变量, $++$ 运算符将 $\text{sizeof}(T)$ 加到该变量的地址中, 而 $--$ 运算符则从该变量的地址中减去 $\text{sizeof}(T)$ 。

如果指针递增或递减运算的结果超过指针类型的域, 则结果是由实现定义的, 但不会产生异常。

18.5.6 指针算法

在不安全上下文中, $+$ 运算符 (§7.7.4) 和 $-$ 运算符 (§7.7.5) 可以应用于除 void^* 以外的所有指针类型的值。因此, 为每个指针类型 T^* 都隐式定义了下列运算符:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

给定指针类型 T^* 的表达式 P 和类型 int 、 uint 、 long 或 ulong 的表达式 N , 表达式 $P + N$ 和 $N + P$ 的计算结果是一个属于类型 T^* 的指针值, 该值等于将 $N * \text{sizeof}(T)$ 加到由 P 给出的地址上。与此类似, $P - N$ 的计算结果也是一个属于类型 T^* 的指针值, 它等于从 P 给出的地址中减去 $N * \text{sizeof}(T)$ 。

给定指针类型 T^* 的两个表达式 P 和 Q , 表达式 $P - Q$ 将先计算 P 和 Q 给出的地址之间的差, 然后用 $\text{sizeof}(T)$ 去除这个差。计算结果的类型始终为 long 。实际上, $P - Q$ 是这样计算的: $((\text{long})(P) - (\text{long})(Q)) / \text{sizeof}(T)$ 。

例如:

```
using System;
class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```


生成以下输出：

```
p - q = -14
q - p = 14
```

如果在执行上述指针算法时，计算结果超越该指针类型的域，则将以实现定义的方式截断结果，但是不会产生异常。

18.5.7 指针比较

在不安全上下文中，`==`，`!=`，`<`，`>`，`<=` 和 `=>` 运算符 (§7.9) 可以应用于所有指针类型的值。指针比较运算符有：

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

由于存在从任何指针类型到 `void*` 类型的隐式转换，因此可以使用这些运算符来比较任何指针类型的操作数。比较运算符像比较无符号整数一样比较两个操作数给出的地址。

18.5.8 sizeof 运算符

`sizeof` 运算符返回由给定类型的变量占用的字节数。被指定为 `sizeof` 的操作数的类型必须是“非托管类型” (§18.2)。

`sizeof-expression:` (`sizeof` 表达式:)

```
sizeof ( unmanaged-type ) (sizeof ( 非托管类型 ))
```

`sizeof` 运算符的结果是 `int` 类型的值。对于某些预定义类型，`sizeof` 运算符将产生如表 18.2 所示的常数值。

表 18.2 sizeof 运算符的结果

表 达 式	结 果
sizeof(sbyte)	1
sizeof(byte)	1
sizeof(short)	2
sizeof(ushort)	2
sizeof(int)	4
sizeof(uint)	4
sizeof(long)	8
sizeof(ulong)	8
sizeof(char)	2
sizeof(float)	4
sizeof(double)	8
sizeof(bool)	1

对于所有其他类型，sizeof 运算符的结果是由实现来定义的，并且属于值而不是常数。一个结构所属的各个成员以什么顺序被装入该结构中，没有明确规定。

出于对齐的目的，在结构的开头、结构内及结构的结尾处可以插入一些未命名的填充位。这些填充位的内容是不确定的。

当 sizeof 应用于具有结构类型的操作数时，结果是该类型变量所占的字节总数（包括所有填充位在内）。

18.6 固定语句

在不安全上下文中，嵌入语句 (§8) 产生式允许使用一个附加结构即 **fixed** 语句，该语句用于“固定”可移动变量，从而使该变量的地址在语句的持续时间内保持不变。

embedded-statement: (嵌入语句:)

...

fixed-statement (固定语句)

fixed-statement: (固定语句:)

fixed (**pointer-type** **fixed-pointer-declarators**) **embedded-statement**
(**fixed** (指针类型 固定指针声明符) 嵌入语句)

fixed-pointer-declarators: (固定指针声明符:)

fixed-pointer-declarator (固定指针声明符)

fixed-pointer-declarators , **fixed-pointer-declarator** (固定指针声明符 , 固定指针声明符)

fixed-pointer-declarator: (固定指针声明符:)

identifier = **fixed-pointer-initializer** (标识符 = 固定指针初始值设定项)

fixed-pointer-initializer: (固定指针初始值设定项:)

& **variable-reference** (& 变量引用)

expression (表达式)

如上述产生式所述，每个固定指针声明符声明一个给定指针类型的局部变量，并使用由相应的固定指针初始值设定项计算的地址初始化该局部变量。在 **fixed** 语句中声明的局部变量的可访问范围仅限于：在该变量声明右边的所有固定指针初始值设定项中，以及在该 **fixed** 语句的嵌入语句中。由 **fixed** 语句声明的局部变量被视为只读。如果嵌入语句试图修改此局部变量（通过赋值或 ++ 和 -- 运算符）或者将它作为 **ref** 或 **out** 参数传递，则会出现编译时错误。

固定指针初始值设定项可以是下列之一：

- “&” 标记，后跟一个变量引用 (§5.4)，它引用非托管类型 **T** 的可移动变量 (§18.3)，前提是类型 **T*** 可以隐式转换为 **fixed** 语句中给出的指针类型。在这种情况下，初始值设定项将计算给定变量的地址，而 **fixed** 语句在生存期内将保证该变量的地址不变。
- 元素类型为 **非托管类型 T** 的数组类型的表达式，前提是类型 **T*** 可隐式转换为

`fixed` 语句中给出的指针类型。在这种情况下，初始值设定项将计算数组中第一个元素的地址，而 `fixed` 语句在生存期内将保证整个数组的地址保持不变。如果数组表达式为 `null` 或者数组具有零个元素，则 `fixed` 语句的行为由实现来定义。

- `string` 类型的表达式，前提是类型 `char*` 可以隐式转换为 `fixed` 语句中给出的指针类型。在这种情况下，初始值设定项将计算字符串中第一个字符的地址，而 `fixed` 语句在生存期内将保证整个字符串的地址不变。如果字符串表达式为 `null`，则 `fixed` 语句的行为是实现定义。

对于每个由固定指针初始值设定项计算的地址，`fixed` 语句确保由该地址引用的变量在 `fixed` 语句的生存期内不会被垃圾回收器重定位或者处置。例如，如果由固定指针初始值设定项计算的地址引用对象的字段或数组实例的元素，`fixed` 语句将保证包含该字段或元素的对象实例本身也不会在该语句的生存期内被重定位或者处置。

确保由 `fixed` 语句创建的指针在执行这些语句之后不再存在是程序员的责任。例如，当 `fixed` 语句创建的指针被传递到外部 API 时，确保 API 不会在内存中保留这些指针是程序员的责任。

固定对象可能导致堆中产生存储碎片（因为它们无法移动）。出于该原因，只有在必要时才应当固定对象，而且固定对象的时间越短越好。

示例：

```
class Test
{
    static int x;
    int y;
    unsafe static void F(int* p) {
        *p = 1;
    }
    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

说明了 `fixed` 语句的几种用法。第一条语句固定并获取一个静态字段的地址，第二条语句固定并获取一个实例字段的地址，第三条语句固定并获取一个数组元素的地址。在这几种情况下，直接使用常规 `&` 运算符都是错误的，这是因为这些变量都属于可移动变量。

上面示例中的第三条和第四条 `fixed` 语句产生相同的结果。一般情况下，对于数组实例 `a`，在 `fixed` 语句中指定 `&a[0]` 与只指定 `a` 等效。

以下是另一个 `fixed` 语句示例，这一次使用 `string`：

```
class Test
{
    static string name = "xx";
    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
```

```

        Console.WriteLine(p[i]);
    }
    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}

```

在不安全上下文中，一维数组的数组元素按递增索引顺序存储，从索引 0 开始，到索引 `Length - 1` 结束。对于多维数组，数组元素按这样的方式存储：首先增加最右边维度的索引，然后是左边紧邻的维度，依此类推直到最左边。在获取指向数组实例 `a` 的指针 `p` 的 `fixed` 语句内，从 `p` 到 `p + a.Length - 1` 范围内的每个指针值均表示数组中的一个元素的地址。与此类似，从 `p[0]` 到 `p[a.Length - 1]` 范围内的变量表示实际的数组元素。已知数组的存储方式，可以将任意维度的数组都视为线性的。

例如：

```

using System;
class Test
{
    static void Main() {
        int[, ,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < 18.Length; ++i) // 视为线性的
                    p[i] = i;
            }
        }
        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
                Console.WriteLine();
            }
    }
}

```

生成以下输出：

```

[0,0,0] = 0 [0,0,1] = 1 [0,0,2] = 2 [0,0,3] = 3
[0,1,0] = 4 [0,1,1] = 5 [0,1,2] = 6 [0,1,3] = 7
[0,2,0] = 8 [0,2,1] = 9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23

```

示例：

```

class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }
    static void Main() {
        int[] a = new int[100];
        unsafe {

```

```

        fixed (int* p = a) Fill(p, 100, -1);
    }
}

```

使用一条 `fixed` 语句固定一个数组，以便可以将该数组的地址传递给一个采用指针作为参数的方法。

通过固定字符串实例产生的 `char*` 类型的值始终指向空终止字符串。在获取指向字符串实例 `s` 的指针 `p` 的 `fixed` 语句内，从 `p` 到 `p + s.Length - 1` 范围内的指针值表示字符串中字符的地址，而指针值 `p + s.Length` 则始终指向一个空字符（值为 `'\0'` 的字符）。

通过固定指针修改托管类型的对象可能导致未定义的行为。例如，由于字符串是不可变的，因此程序员应确保指向固定字符串的指针所引用的字符不被修改。

这种字符串的自动空终止功能，大大方便了调用需要“C 风格”字符串的外部 API。但请注意，核心 C# 允许字符串实例包含空字符。如果字符串中存在此类空字符，则在将字符串视为空终止的 `char*` 时会出现截断。

18.7 堆栈分配

在不安全上下文中，局部变量声明 (§8.5.1) 可以包含一个从调用堆栈中分配内存的堆栈分配初始值设定项。

local-variable-initializer: (局部变量初始值设定项:)

expression (表达式)

array-initializer (数组初始值设置项)

stackalloc-initializer (**stackalloc** 初始值设置项)

stackalloc-initializer: (**stackalloc** 初始值设定项:)

stackalloc **unmanaged-type** [**expression**] (**stackalloc** 非托管类型 [表达式])

在上述产生式中，非托管类型标识将在新分配的位置中存储的项的类型，而表达式则标识这些项的数目。合在一起，它们指定所需的分配大小。由于堆栈分配的大小不能为负值，因此将项的数目指定为计算结果为负值的常数表达式会导致编译时错误。

`stackalloc T[E]` 形式的堆栈分配初始值设定项要求：`T` 必须为非托管类型 (§18.2)，`E` 必须为 `int` 类型的表达式。该构造从调用堆栈中分配 `E * sizeof(T)` 个字节，并返回一个指向新分配的块的、类型 `T*` 的指针。如果 `E` 为负值，则其行为是未定义的。如果 `E` 为零，则不进行任何分配，并且返回的指针由实现来定义。如果没有足够的内存分配给定大小的块，则抛出 `System.StackOverflowException`。

新分配的内存的内容是未定义的。

在 `catch` 或 `finally` 块 (§18.10) 中不允许使用堆栈分配初始值设定项。

无法显式释放利用 `stackalloc` 分配的内存。在函数成员的执行期间创建的所有堆栈分配内存块都将在该函数成员返回时自动丢弃。这对应于 `alloca` 函数，它是通常存在于 C 和 C++ 实现中的一个扩展。

示例:

```
using System;
class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }
    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

在 `IntToString` 方法中使用了 `stackalloc` 初始值设定项，以在堆栈上分配一个 16 个字符的缓冲区。此缓冲区在该方法返回时被自动丢弃。

18.8 动态内存分配

除 `stackalloc` 运算符外，C# 不提供其他预定义构造来管理那些不受垃圾回收控制的内存。这些服务通常是由支持类库提供或者直接从底层操作系统导入的。例如，下面的 `Memory` 类阐释了可以如何从 C# 访问底层操作系统的有关堆处理的各种功能：

```
using System;
using System.Runtime.InteropServices;
public unsafe class Memory
{
    // 进程堆的句柄。该句柄用于下面的方法中所有对 HeapXXX APIs 的调用。
    static int ph = GetProcessHeap();
    // 私有实例构造函数，以防止初始化。
    private Memory() {}
    // 分配给定大小的内存块。该被分配的内存被自动初始化为零。
    public static void* Alloc(int size) {
        void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }
    // 由 src 到 dst 拷贝 count 个字节。允许源块和目标块重叠。
    public static void Copy(void* src, void* dst, int count) {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd) {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd) {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
        }
    }
}
```

```

    }
}
// 释放一块内存。
public static void Free(void* block) {
    if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
}
// 重新分配一块内存。如果重新分配的内存块更大，则内存多出的区域自动初始化为零。
public static void* ReAlloc(void* block, int size) {
    void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
    if (result == null) throw new OutOfMemoryException();
    return result;
}
// 返回内存块的大小。
public static int SizeOf(void* block) {
    int result = HeapSize(ph, 0, block);
    if (result == -1) throw new InvalidOperationException();
    return result;
}
// Heap API 标记
const int HEAP_ZERO_MEMORY = 0x00000008;
// Heap API 函数
[DllImport("kernel32")]
static extern int GetProcessHeap();
[DllImport("kernel32")]
static extern void* HeapAlloc(int hHeap, int flags, int size);
[DllImport("kernel32")]
static extern bool HeapFree(int hHeap, int flags, void* block);
[DllImport("kernel32")]
static extern void* HeapReAlloc(int hHeap, int flags,
    void* block, int size);
[DllImport("kernel32")]
static extern int HeapSize(int hHeap, int flags, void* block);
}

```

以下给出一个使用 **Memory** 类的示例：

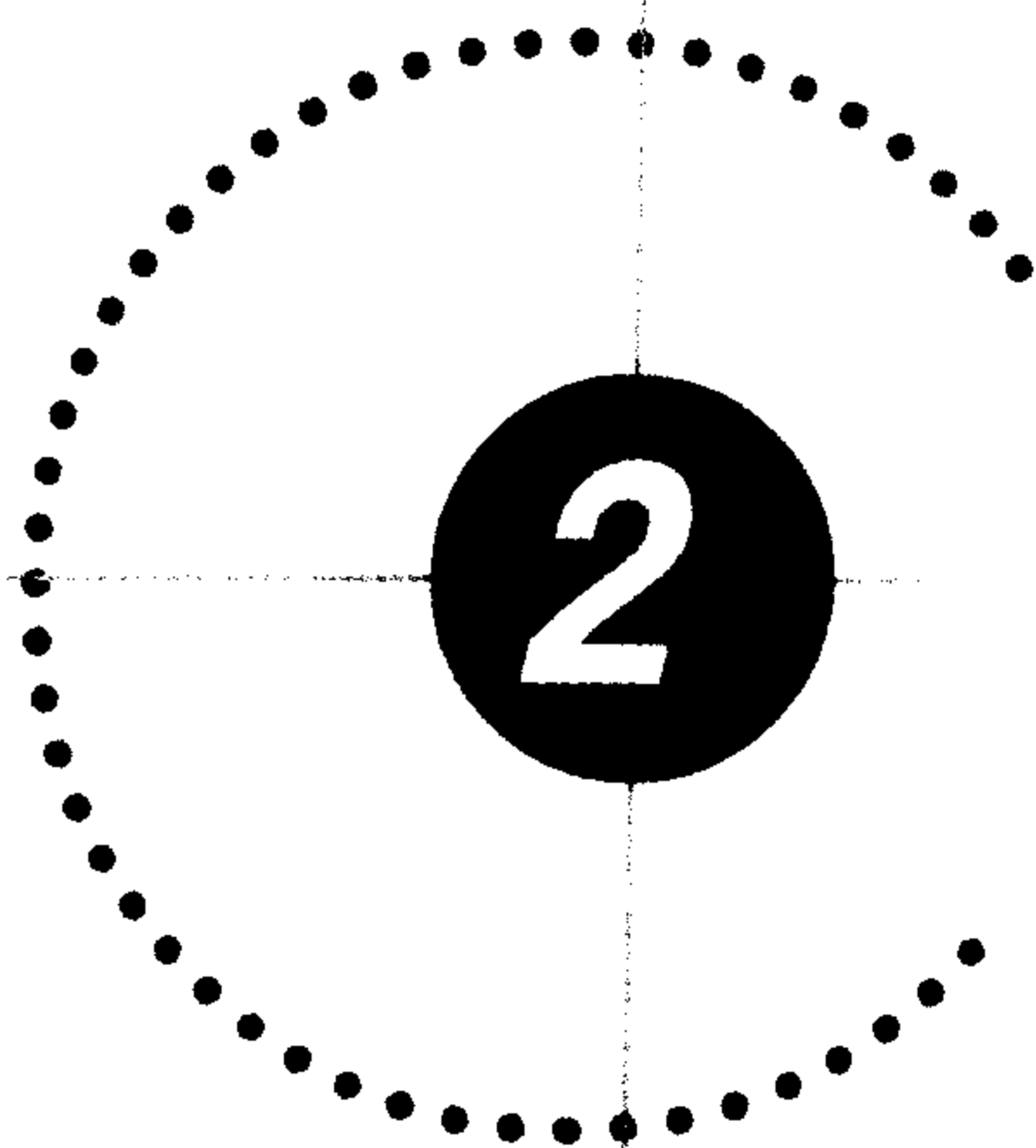
```

class Test
{
    static void Main() {
        unsafe {
            byte* buffer = (byte*)Memory.Alloc(256);
            try {
                for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
                byte[] array = new byte[256];
                fixed (byte* p = array) Memory.Copy(buffer, p, 256);
            }
            finally {
                Memory.Free(buffer);
            }
            for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
        }
    }
}

```

此示例通过 **Memory.Alloc** 分配了 256 字节的内存，并且使用从 0 增加到 255 的值初始化该内存块。它随后分配一个具有 256 个元素的字节数组并使用 **Memory.Copy** 将内存块的内容复制到此字节数组中。最后，使用 **Memory.Free** 释放内存块，并将字节数组的内容输出到控制台上。

第二篇 C# 2.0



第 19 章 C# 2.0 介绍

C# 2.0 引入了几项语言扩展，其中最重要的是泛型 (generics)、匿名方法 (anonymous methods)、迭代器 (iterators) 和不完整类型 (partial type)。

- 泛型可以让类、结构、接口、委托和方法通过它们所存储和操纵的数据的类型被参数化。泛型是很有用的，因为它们提供了更强的编译时类型检查，减少了数据类型之间的显式转换，以及装箱操作和运行时类型检查。
- 匿名方法可以让代码块以内联的方式潜入到期望委托值的地方。匿名方法与 Lisp 编程语言中的 λ 函数 (lambda function) 相似。C# 2.0 支持 “closures” 的创建，在其中匿名方法可以访问相关局部变量和参数。
- 迭代器是可以递增计算和产生值的方法。迭代器让类型指定 foreach 语句如何迭代它的所有元素变得很容易。
- 不完整类型可以让类、结构和接口被拆分成多个部分存储在不同的源文件中，这更利于开发和维护。此外，不完整类型允许某些类型的机器生成的部分与用户编写的部分之间的分离，从而使增加由工具产生的代码很容易。

本章将介绍这些新特征。介绍完之后，接下来的 4 章提供了这些特征的完整的技术规范。

C# 2.0 的语言扩展主要被设计用于确保与现存的代码之间最大的兼容性。例如，尽管 C# 2.0 对于 where, yield 和 partial 这些词在特定上下文中赋予了特别的意义，但这些词仍然可用做标识符。实际上，C# 2.0 没有增加任何可能与现有代码中的标识符冲突的关键字。

19.1 泛型

泛型可以让类、结构、接口、委托和方法通过它们所存储和操纵的数据的类型被参数化。C# 泛型对于使用 Eiffel 或 Ada 的泛型的用户，或者对于 C++ 模板的用户来说是很熟悉的，但他们将不用再去忍受后者众多的复杂性。

19.1.1 为什么使用泛型

如果没有泛型，通用目的的数据结构可以采用 object 类型存储任何类型的数据。例如，下面的 Stack 类在一个 object 数组中存储数据，而它的两个方法 (Push 和 Pop) 相应地使用 object 接收和返回数据。

```
public class Stack
{
    object[] items;
```

```

    int count;
    public void Push(object item){...}
    public object Pop(){...}
}

```

尽管使用类型 `object` 可以使 `Stack` 类更加灵活，但这样做也并不是没有缺点。例如，你可以将一个任何类型的值（诸如 `Customer` 的一个实例）压入（`Push`）堆栈。但当你取回一个值时，`Pop` 方法的结果必须被显式地强制转换为合适的类型，为一个运行时类型检查去编写代码，以及带来的性能不利影响，是很令人讨厌的：

```

Stack stack = new Stack();
Stack.Push(new Customer());
Customer c = (Customer)stack.Pop();

```

如果一个值类型的值（例如一个 `int`）被传递到 `Push` 方法，那么它将被自动装箱。当后面获得这个 `int` 时，它必须使用一个显式的强制转换而被取消装箱：

```

Stack stack = new Stack();
Stack.Push(3);
int i = (int)stack.Pop();

```

这种装箱和取消装操作增加了性能开销，因为它们涉及动态内存分配和运行时类型检查。

`Stack` 类的更大的问题是，它不能强制放置在堆栈上的数据种类。实际上，`Customer` 实例可以被压入堆栈，而取回它时可能被强制转换到错误的类型：

```

Stack stack = new Stack();
Stack.Push(new Customer());
String s = (string)stack.Pop();

```

尽管上面的代码是 `Stack` 类的一种不恰当用法，但这段代码从技术上说是正确的，并且也不会报告编译时错误。问题直到代码执行时才会冒出来，在这一点上将会抛出一个 `InvalidCastException` 异常。

如果 `Stack` 类具有指定其元素类型的能力，那么很显然它能从这种能力得到好处。使用泛型，这将会变得可能。

19.1.2 创建和使用泛型

泛型为创建具有类型参数（`type parameter`）的类型提供了工具。下面的例子声明了一个带有类型参数 `T` 的泛型 `Stack` 类。类型参数在类名字之后的“<”和“>”分界符中指定。这里没有 `object` 与别的类型之间的相互转换，`Stack<T>` 的实例接受它们被创建时的类型，并且存储那个类型的数据而不转换它。类型参数 `T` 充当一个占位符，直到使用的时候才指定一个实际的类型。注意，`T` 用做内部 `items` 数组的元素类型、`Push` 方法参数的类型和 `Pop` 方法的返回值类型。

```

Public class Stack<T>
{
    T[] items;
    int count;
}

```

```

        public void Push(T item){...}
        public T Pop(){...}
    }

```

当泛型类 `Stack<T>` 被使用时, `T` 所代替的实际类型将被指定。在下面的例子中, `int` 将作为 `T` 的类型参数给出。

```

Stack<int> stack = new Stack<int>();
Stack.Push(3);
int x = stack.Pop();

```

`Stack<int>` 类型被称为构造类型 (constructed type)。在 `Stack<int>` 类型中, `T` 的每次出现都被使用类型参数 `int` 代替。当 `Stack<int>` 的实例被创建时, `items` 数组的本地存储就是一个 `int[]` 而不是 `object[]`, 与非泛型 `Stack` 相比, 它提供了更高的存储效率。同样地, 在 `int` 值上的 `Stack<int>` 操作的 `Push` 和 `Pop` 方法, 将会使压入其他类型的值到堆栈中时出现一个编译时错误, 并且当取返回值的时候也不需要转换回它们原始的类型。

泛型提供了强类型, 意味着例如压入一个 `int` 到 `Customer` 对象堆栈都会出现错误。就好像 `Stack<int>` 被限制为只能在 `int` 值上操作一样, `Stack<Customer>` 也被限制用于 `Customer` 对象。

对于下面的例子, 编译器将会在最后两行报告错误。

```

Stack<Customer> stack = new Stack<Customer>();
Stack.Push(new Customer());
Customer c = stack.Pop();
stack.Push(3);           //类型不匹配错误
int x = stack.Pop();     //类型不匹配错误

```

泛型类型声明可以有任意数量的类型参数。前面的 `Stack<T>` 例子只有一个类型参数, 但一个通用的 `Dictionary` 类可能有两个类型参数: 一个用于键 (key) 的类型, 另一个用于值 (value) 的类型。

```

public class Dictionary<K , V>
{
    public void Add(K key , V value){...}
    public V this[K key]{...}
}

```

当 `Dictionary<K, V>` 被使用时, 必须提供两个类型参数:

```

Dictionary<string , Customer> dict = new Dictionary<string , Customer>();
dict.Add("Peter", new Customer());
Customer c = dict["Peter"];

```

19.1.3 泛型类型实例化

与非泛型类型相似, 被编译过的泛型类型也是由中间语言 (Intermediate Language, IL) 指令和元数据表示的。泛型类型的表示当然也对类型参数的存在和使用进行了编码。

当应用程序首次创建一个构造泛型类型的实例 (例如 `Stack<int>`) 时, .NET 公共语言运行时的实时编译器 (JIT) 将在进程中把泛型 IL 和元数据转换为本地代码, 并且将类型参数替换为实际的类型。对于那个构造泛型类型的后续引用将会使用相同的本机代码。从

一个泛型类型创建一个特定构造类型的过程，称为**泛型类型实例化**（**generic type instantiation**）。

.NET 公共语言运行时使用值类型为每个泛型类型实例创建了一个本地代码的特定拷贝，但对于所有的引用类型，它将共享那份本地代码的单一拷贝（因为，在本地代码级别，引用只是带有相同表示的指针）。

19.1.4 约束

一般来讲，泛型类不限于只是根据类型参数存储值。泛型类经常可能在给定类型参数的类型的对象上调用方法。例如，`Dictionary<K, V>`类中的 `Add` 方法可能需要使用 `CompareTo` 方法比较键值。

```
public class Dictionary<K,V>
{
    public void Add(K key , V value)
    {
        ...
        if(key.CompareTo(x)<0){...} //错误，没有 CompareTo 方法
        ...
    }
}
```

因为为 `K` 所指定的类型参数可能是任何类型（可以假定 `key` 参数存在的惟一成员，就是那些被声明为 `object` 类型的，例如，`Equals`，`GetHashCode` 和 `ToString`），因此，在先前例子中将会出现编译时错误。当然，你可以将 `key` 参数强制转换到一个包含 `CompareTo` 方法的类型。例如，`key` 参数可能被强制转换到 `IComparable` 接口。

```
public class Dictionary<K , V>
{
    public void Add(K key , V value)
    {
        ...
        if(((IComparable)key).CompareTo(x)<0){...}
        ...
    }
}
```

尽管这种解决办法有效，但它需要在运行时的动态类型检查，这也增加了开销。更糟糕的是，它将错误报告推迟到了运行时，如果键没有实现 `IComparable` 接口将会抛出 `InvalidCastException` 异常。

为了提供更强的编译时类型检查，并减少类型强制转换，C#允许为每个类型参数提供一个**约束**（**constraint**）的可选列表。类型参数约束指定了类型必须履行的一种需求，其目的是为了该类型参数用做实参（**argument**）。约束使用单词 `where` 声明，随后是类型参数的名字，接着是类或接口类型的列表，以及可选的构造函数约束 `new()`。

```
public class Dictionary<K, V> where K :IComparable
{
    public void Add(K key , V value)
    {
        ...
    }
}
```

```

        if (key.CompareTo(x) < 0) {...}
        ...
    }
}

```

给定这个声明, 编译器将会确保 **K** 的任何类型实参是实现了 **Comparable** 接口的类型。

并且, 在调用 **CompareTo** 方法之前也不再需要对 **key** 参数进行显式的强制转换。为类型参数作为一个约束而给出的类型的所有成员, 对于类型参数类型的值是直接有效的。

对于一个给定的类型参数, 你可以指定任意数量的接口作为约束, 但只能有一个类。每个约束的类型参数有一个单独的 **where** 语句。在下面的例子中, 类型参数 **K** 有两个接口约束, 类型参数 **E** 有一个类约束和一个构造函数约束。

```

public class EntityTable<K, E>
    where K:Comparable<K>,IPersistable
    where E:Entity, new()
{
    public void Add(K key , E entity)
    {
        ...
        if (key.CompareTo(x) < 0) {...}
        ...
    }
}

```

在上面的例子中, 构造函数约束 **new()**, 确保为使 **E** 用做类型参数的类型具有一个公有的、无参数构造函数, 并且它允许泛型类使用 **new E()** 创建该类型的实例。

对于类型参数约束应该很小心地使用。尽管它们提供了更强的编译时类型检查, 在某些情况下增强了性能, 但它们也限制了泛型类型的可能的用法。例如, 泛型类 **List<T>** 可能约束 **T** 实现 **Comparable** 接口, 由此它的 **Sort** 方法可以比较项的大小。然而, 这样做却导致没有实现 **Comparable** 接口的类型不能使用 **List<T>** (即使在这些情形下, **Sort** 方法根本就没有被调用过)。

19.1.5 泛型方法

在某些情形下, 类型参数对于整个类不是必需的, 而只在特定方法内是必需的。在创建一个接受泛型类型作为参数的方法时常常就是这样。例如, 当使用早些时候描述的 **Stack<T>** 类时, 通用的模式可能是在一行中压入多个值, 在一个单一的调用中写一个方法。这么做也是很方便的。对于特定的构造类型, 例如 **Stack<int>**, 这个方法如下所示:

```

void PushMultiple(Stack<int> stack ,params int[] values)
{
    foreach(int value in values)
        stack.Push(value);
}

```

这个方法可以用于将多个 **int** 值压入到一个 **Stack<int>** 中。

```

Stack<int> stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);

```


然而，上面的方法只对于特定的构造类型 `Stack<int>` 有效。要让它对于 `Stack<T>` 也起作用，方法必须作为泛型方法（generic method）而编写。泛型方法在方法的名字后面在“<”和“>”分界符之间指定了一个或多个类型参数。类型参数可以在参数列表、返回类型和方法体之内被使用。一个泛型的 `PushMultiple` 方法如下所示：

```
void PushMultiple<T>(Stack<T> stack , params T[] values)
{
    foreach(T value in values) stack.Push(value);
}
```

使用这个泛型方法，你可以将多个项压入任意 `Stack<T>` 中。当调用一个泛型方法时，类型参数值在方法调用的尖括号中给定。例如：

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack , 1,2,3,4);
```

这个泛型 `PushMultiple` 方法比先前的版本更具有重用性，因为它可以工作在任何 `Stack<T>` 上，但似乎在调用的时候不太方便，因为必须提供 `T` 作为一个类型参数传递给方法。在许多情形下，编译器使用一种称为**类型推断**（**type inferencing**）的处理过程，从传递给方法的其他参数推断正确的类型参数。在上面的例子中，因为第一个正式参数是 `Stack<int>` 类型，而后续的参数是 `int` 类型，因此编译器可以推断类型参数值必须是 `int`。因此，在调用泛型 `PushMultiple` 方法时可以不指定类型参数。

```
Stack<int> stack = new Stack<int>();
PushMultiple(stack , 1,2, 3, 4);
```

19.2 匿名方法

事件句柄和其他回调函数经常需要通过专门的委托调用，从来都不是直接调用。虽然如此，我们还是只能将事件句柄和回调函数的代码放在特定方法中，再显式地为这个方法创建委托。相反，**匿名方法**（**anonymous method**）允许一个委托关联地代码被内联地写入使用委托的地方，很方便的是这使得代码对于委托的实例很直接。除了这种便利之外，匿名方法还共享了对本地语句包含的函数成员的访问。为了使命名方法实现共享（区别于匿名方法），需要手工创建辅助类，并将本地成员“提升（lifting）”为类的域。

下面的例子展示了一个简单的输入表单，它包含一个列表框、一个文本框和一个按钮。当按钮被按下时，在文本框中一个包含文本的项就被添加到列表框。

```
class InputForm:Form
{
    ListBox listBox;
    TextBox textbox;
    Button addButton;
    public MyForm()
    {
        listBox = new ListBox(...);
        textbox = new TextBox(...);
        addButton = new Button(...);
        addButton.Click += new EventHandler(AddClick);
    }
}
```

```

void AddClick(object sender , EventArgs e)
{
    listBox.Items.Add(textbox.Text);
}
}

```

即使作为对按钮的 Click 事件的响应只有惟一的一条语句需要执行，那条语句也必须放在一个具有完整的参数列表的单独的方法中，并且还必须手工创建引用那个方法的 EventHandler 委托。使用匿名方法，事件处理代码将变得相当简洁：

```

class InputForm:Form
{
    ListBox listBox;
    TextBox textBox;
    Button addButton;
    public MyForm()
    {
        listBox = new ListBox(...);
        textBox = new TextBox(...);
        addButton = new Button(...);
        addButton.Click +=delegate{
            listBox.Items.Add(textBox.Text);
        };
    }
}

```

匿名方法由关键词 delegate 和一个可选的参数列表，以及一个封闭在“{”和“}”分界符中的语句组成。前面的例子中匿名方法没有使用由委托所提供的参数，所以便省略了参数列表。如果要访问参数，匿名方法可以包含一个参数列表。

```

addButton.Click += delegate(object sender , EventArgs e){
    MessageBox.Show(((Button)sender).Text);
};

```

在上面的例子中，将会发生一次从匿名方法到 EventHandler 委托类型（Click 事件的类型）的隐式转换。这种隐式转换是可能的，因为参数列表和委托类型的返回值与匿名方法是兼容的。关于兼容性的确切规则如下：

- 如果下列之一成立，那么委托的参数列表与匿名方法是兼容的。
 - 匿名方法没有参数列表，并且委托没有 out 参数。
 - 匿名方法包含的参数列表与委托的参数在数量、类型和修饰符上是精确匹配的。
- 如果下列之一成立，那么委托的返回类型与匿名方法兼容。
 - 委托的返回类型是 void，匿名方法没有返回语句，或者只有不带表达式的 return 语句。
 - 委托的返回类型不是 void，并且在匿名方法中，所有 return 语句相关的表达式可以被隐式转换到委托的类型。

在委托类型的隐式转换发生以前，委托的参数列表和返回类型二者都必须与匿名方法兼容。

下面的例子使用匿名方法编写了“内联”函数。匿名方法作为 Function 委托类型而传递。

```

using System;

```

```

delegate double Function(double x);
class Test
{
    static double[] Apply(double[] a ,Function f) {
        double[] result = new double[a.Length];
        for(int i=0;i<a.Length;i++) result[i] = f(a[i]);
        return result;
    }
    static double[] MultiplyAllBy(double[] a, double factor){
        return Apply(a, delegate(double x){return x * factor;})
    }
    static void Main(){
        double[] a = {0.0,0.5,1.0};
        double[] squares = Apply(a, delegate(double x){return x * x});
        double[] doubles = MultiplyAllBy(a , 2.0);
    }
}

```

Apply 方法应用 `double[]` 元素的给定的 **Function**，并返回一个 `double[]` 作为结果。在 **Main** 方法中，传递给 **Apply** 的第二个参数是一个匿名方法，它与 **Function** 委托类型兼容。该匿名方法只是返回参数的平方，而 **Apply** 调用的结果是一个 `double[]`，在 `a` 中包含了值的平方。

MultiplyAllBy 方法返回一个通过给定 **factor**（因数）与在参数数组 `a` 中的每个值相乘而创建的 `double[]`。为了得到结果，**MultiplyAllBy** 调用了 **Apply** 方法，并传给它一个匿名方法（在参数上乘以因数 **factor**）。

如果一个本地变量或参数的作用域包括了匿名方法，则该变量和参数被称为匿名方法的外部变量（**outer variable**）。在 **MultiplyAllBy** 方法中，`a` 和 `factor` 是传递给 **Apply** 的匿名方法的外部变量，因为匿名方法引用了 `factor`，`factor` 被匿名方法所捕获（**capture**）。通常，局部变量的生存期被限制在它所关联的块或语句的执行区。然而，被捕获的外部变量将一直存活到委托所引用的匿名方法可以被垃圾回收为止。

如前面所描述的，匿名方法可以被隐式地转换到与之兼容的委托类型。对于一个方法组，C# 2.0 允许这种相同类型的转换，即在几乎任何情况下都不需要显式的实例化委托。例如，下面的语句：

```

addButton.Click += new EventHandler(AddClick);
Apply(a , new Function(Math.Sin));

```

可以被如下语句代替：

```

addButton.Click += AddClick;
Apply(a , Math.Sin);

```

当使用这种简短的形式时，编译器将自动推断哪一个委托类型需要实例化，但其最后的效果与较长的表达形式是一样的。

19.3 迭代器

C#的 `foreach` 语句被用于迭代一个可枚举（**enumerable**）集合的所有元素。为了可以

被枚举，集合必须具有一个无参数 `GetEnumerator` 方法，它返回一个 **enumerator** (枚举器)。一般情况下，枚举器是很难实现的，但这个问题使用迭代器就大大地简化了。

迭代器是产生值的有序序列的一个语句块。迭代器不同于有一个或多个 `yield` 语句存在的常规语句块。

- `yield return` 语句产生迭代的下一个值。
- `yield break` 语句指明迭代已经完成。

只要函数成员的返回类型是枚举器接口 (`enumerator interface`) 或可枚举接口 (`enumerable interface`) 之一，迭代器就可以被用做函数体。

- 枚举器接口是 `System.Collections.IEnumerator` 和由 `System.Collections.Generic.IEnumerator<T>` 所构造的类型。
- 可枚举接口是 `System.Collections.IEnumerable` 和由 `System.Collections.Generic.IEnumerable<T>` 构造的类型。

迭代器不是一种成员，它只是实现函数成员的方式，理解这一点是很重要的。一个通过迭代器被实现的成员，可以被其他可能或不可能通过迭代器而被实现的成员覆盖和重载。

下面的 `Stack<T>` 类使用迭代器实现了它的 `GetEnumerator` 方法。这个迭代器依序枚举了堆栈中从顶到底的所有元素。

```
using System.Collections.Generic;
public class Stack<T>:IEnumerable<T>
{
    T[] items;
    int count;
    public void Push(T data){...}
    public T Pop(){...}
    public IEnumerator<T> GetEnumerator() {
        for(int i =count-1;i>=0;--i){
            yield return items[i];
        }
    }
}
```

`GetEnumerator` 方法的存在使得 `Stack<T>` 成为一个可枚举类型，它使得 `Stack<T>` 的实例可被用在 `foreach` 语句中。下面的例子将从 0 到 9 的值压入一个整数堆栈中，并且使用一个 `foreach` 循环依序显示堆栈中从顶到底的所有值。

```
using System;
class Test
{
    static void Main() {
        Stack<int> stack = new Stack<int>();
        for(int i=0;i<10;i++) stack.Push(i);
        foreach(int i in stack) Console.Write("{0}",i);
        Console.WriteLine();
    }
}
```

该例的输出如下：

9 8 7 6 5 4 3 2 1 0

`foreach` 语句隐式地调用了集合的无参数 `GetEnumerator` 方法以获得一个枚举器。由集

合所定义的只能有一个这样的无参数 GetEnumerator 方法，但经常有多种枚举方式，以及通过参数控制枚举的方法。在这种情况下，集合可以使用迭代器实现返回可枚举接口之一的属性和方法。例如，Stack<T>可能引入两个 IEnumerable<T>类型的新属性：TopToBottom 和 BottomToTop。

```
using System.Collections.Generic;
public class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;
    public void Push(T data){...}
    public T Pop(){...}
    public IEnumerable<T> GetEnumerator() {
        for(int i= count-1;i>=0;--i) {
            yield return items[i];
        }
    }
    public IEnumerable<T> TopToBottom{
        get{
            return this;
        }
    }
    public IEnumerable<T> BottomToTop{
        get{
            for(int i = 0;i<count;i++) {
                yield return items[i];
            }
        }
    }
}
```

TopToBottom 属性的 get 访问器只是返回 this，因为堆栈自身是可枚举的。BottomToTop 属性返回一个使用 C#迭代器实现的枚举。下面的例子展示了属性如何用于枚举堆栈元素：

```
using System;
class Test
{
    static void Main() {
        Stack<int> stack = new Stack<int>();
        for(int i = 0 ;i<10 ;i++) stack.Push(i);

        for each (int i in stack.TopToBottom) Console.Write("{0}",i);
        Console.WriteLine();

        foreach (int i in stack.BottomToTop) Console.Write("{0}",i);
        Console.WriteLine();
    }
}
```

当然，这些属性同样也可以在 foreach 语句之外使用。下面的例子将调用属性的结果传递给了一个单独的 Print 方法。该例子也展示了一个用做 FromToBy 接受参数的方法体的迭代器：

```
using System;
using System.Collections.Generic;
class Test
{
    static void Print(IEnumerable<int> collection) {
```

```

        foreach(int i in collection) Console.Write("{0}",i);
        Console.WriteLine();
    }
    static IEnumerable<int> FromToBy(int from ,int to , int by) {
        for(int i =from ;i<=to ;i +=by) {
            yield return i;
        }
    }
    static void Main() {
        Stack<int> stack = new Stack<int>();
        for(int i= 0 ;i<10;i ++) stack.Push(i);
        Print(stack.TopToBottom);
        Print(stack.BottomToTop);
        Print(FromToBy(10,20,2));
    }
}

```

该例子的输出如下:

```

9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
10 12 14 16 18 20

```

泛型和非泛型可枚举接口包含一个单一的成员, 一个不接受参数的 `GetEnumerator` 方法, 并且返回一个枚举器接口。一个枚举充当一个枚举器工厂。每当调用了一个正确地实现了可枚举接口的类的 `GetEnumerator` 方法时, 都会产生一个独立的枚举器。假定枚举的内部状态在两次调用 `GetEnumerator` 之间没有改变, 返回的枚举器应该产生相同集合、相同顺序的枚举值。在下面的例子中, 即使枚举的生存期发生交叠, 这一点也应该保持。

```

using System;
using System.Collections.Generic;
class Test
{
    static IEnumerable<int> FromTo(int from , int to) {
        while(from<=to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1,10);
        foreach(int x in e) {
            foreach(int y in e) {
                Console.Write (" {0,3}",x*y);
            }
            Console.WriteLine();
        }
    }
}

```

上面的代码打印了整数 1 到 10 的乘法表。注意, `FromTo` 方法只被调用了一次用来产生枚举 `e`。然而, `e.GetEnumerator()` 被调用了多次 (通过 `foreach` 语句), 以产生多个等价的枚举器。这些枚举器都封装了在 `FromTo` 声明中指定的迭代器代码。注意迭代器代码修改了 `from` 参数。

不过, 枚举器是独立地运作的, 因为每个枚举器都给出 `from` 和 `to` 参数自己的拷贝。枚举器之间过渡状态的共享是众多细微的瑕疵之一, 当实现枚举和枚举器时应该避免。C# 迭代器的设计可用于避免这些问题, 从而以一种简单直观的方式实现健壮的枚举和枚举器。

19.4 不完整类型

尽管在一个单一的文件中，为一个类型维护所有的源代码是一个良好的编程实践，但有时一个类型变得非常大，这将成为一个不切实际的限制。此外，程序员经常使用源代码生成器产生应用程序的初始结构，并且修改结果代码。遗憾的是，当源代码在将来被再次发射时，现存的修改将会被覆盖。

不完整类型 (partial type) 可以让类、结构和接口被拆分成多个部分存储在不同的源文件中，这更利于开发和维护。此外，不完整类型允许某些类型的机器生成的部分与用户编写的部分之间的分离，因此增加由工具产生的代码很容易。

当在多个部分中定义一个类型时，你可以使用一个新的类型修饰符 `partial`。下面是一个不完整类的例子，它是在两个部分中实现的。这两个部分可以在不同的源文件中，例如，当第一个部分是通过一个数据库映射工具由机器生成的，第二个部分是由手工创建的时。

```
public partial class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Order> orders;
    public Customer() {
        ...
    }
}

public partial class Customer
{
    public void SubmitOrder(Order order) {
        orders.Add(order);
    }
    public bool HasOutstandingOrders() {
        return orders.Count>0;
    }
}
```

当前面的两个部分一起编译时，其结果代码和被作为一个单一的单元而编写的类是一样的。

```
public class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Order> orders;
    public Customer() {
        ...
    }

    public void SubmitOrder(Order order) {
        orders.Add(order);
    }
    public bool HasOutstandingOrders() {
        return orders.Count>0;
    }
}
```


不完整类型的所有部分必须一起编译，这样各个部分在编译时就可以被融合到一起。特别需要注意的是，不完整类型不允许对已经被编译的类型进行扩展。

第 20 章 泛型

20.1 泛型类声明

泛型类声明是一种类声明，它需要提供类型参数以形成实际类型。

类声明可以有选择地定义类型参数。

class-declaration: (类声明)

attributes_{opt} class-modifiers_{opt} class identifier type-parameter-list_{opt} class
-base_{opt} type-parameter-constraints-clauses_{opt} class-body;_{opt} (特性_{可选} 类修饰符_{可选}
类标识符_{可选} 类型参数列表_{可选} 基类_{可选} 类型参数约束语句_{可选} 类体;_{可选})

只有提供了类型参数列表，这个类声明才可以提供类型参数化约束语句。

提供了类型参数列表的类声明是一个泛型类声明。此外，任何嵌入到泛型类声明或泛型结构声明中的类，自身是一个泛型类声明，因为必须提供包含类型的类型参数以创建构造类型 (constructed type);

泛型类通过使用构造类型而被引用 (§ 20.5)。给定泛型类声明

```
class List<T>{}
```

这是构造类型的一些例子，List<T>，List<int>和 List<List<string>>。构造类型可以使用一个或多个参数，例如 List<T>，称为**开放构造类型 (open constructed type)**。不使用类型参数的构造类型，例如 List<int>，称为**封闭构造类型 (closed constructed type)**。

泛型类型不可以被“重载”。也就是说，和普通类型一样，在一个作用域内，泛型类型必须被唯一地命名。

```
class C{}  
class C<V>{} //错误, C 定义了两次  
class C<U,V>{} //错误, C 定义了两次
```

然而在非限定类型名字查找 (§ 20.9.3) 中使用的类型查找规则和成员访问 (§ 20.9.4)，确实考虑到了类型参数的个数。

20.1.1 类型参数

类型参数可以在一个类声明上提供。每个类型参数是一个简单的标识符，它指示了用来创建一个构造类型的类型参数的占位符。类型参数是在后面将要被提供的类型的形式占

位符。相反，类型实参 (§ 20.5.1)^{注 1}才是实际的类型，在构造类型被引用时，它是类型参数的一个替代。

type-parameter-list: (类型参数列表:)

<type-parameters> (<类型参数>)

type-parameters: (类型参数:)

type-parameter (类型参数)

type-parameters, type-parameter (类型参数, 类型参数)

type-parameter: (类型参数:)

attributes_{opt} identifier (特性_{可选} 标识符)

类声明中的每个类型参数在类的声明空间 (§ 3.3) 定义了一个名字。由此，它不能和另一个类型参数或在类中声明的成员具有同样的名字。类型参数不能和类型自身有同样的名字。

在类中一个类型参数的作用域 (§ 3.7) 包括基类、类型参数约束语句和类体。与类的成员不同的是，它没有扩展到派生类。在其作用域之内，类型参数可以用做一个类型。

Type: (类型):

value-type (值类型)

reference-type (引用类型)

type-parameter (类型参数)

由于类型参数可以被许多不同的实际类型实参所实例化，因此类型参数与其他类型相比将略微有一些不同的操作和限制。包括如下内容：

- 类型参数不能用于直接声明一个基类型或者接口。
- 对于在类型参数上的成员查找规则，如果约束存在，则依赖于应用到该类型参数的约束。更详细的说明参看 § 20.7.2。
- 有效的类型参数转换依赖于应用到该类型参数上的约束（如果有的话）。详细的说明参看 § 20.7.4。
- 文本 `null` 不能被转换到由类型参数所给定的类型，除非类型参数是由一个类约束 (§ 20.7.4) 所约束的。但可以使用一个默认值表达式 (§ 20.8.1) 代替。此外，由一个类型参数给定的类型的值可以使用 “==” 和 “!=” (§ 20.8.4) 与 `null` 进行比较。
- 如果类型参数通过一个构造函数约束 (constructor-constraint) (§ 20.7) 被约束，则 `new` 表达式 (§ 20.8.2) 只能为类型参数所使用。
- 类型参数不能用于特性内的任何地方。
- 类型参数不能用于成员访问，或者标识一个静态成员或者嵌套类型的类型名字 (§ 20.9.1, § 20.9.4)。
- 在不安全代码中，类型参数不能用做非托管类型 (§ 18.2)。

作为一种类型，类型参数纯粹只是一个编译时构件。在运行时，每个类型参数被绑定

^{注 1} Type Argument 译为类型实参，表示实际的类型，Type Parameter 译为类型参数，只表示一个形式占位符。

到运行时类型，这是通过泛型类型声明所提供的类型实参而指定的。为此，在运行时，使用类型参数声明的变量类型是一个封闭类型（closed type）（§ 20.5.2）。所有语句和表达式在运行时执行所使用的类型参数，都是由那个参数作为类型实参而提供的实际类型。

20.1.2 实例类型

每个类声明都有与之关联的构造类型，即**实例类型（instance type）**。对于一个泛型类声明，实例类型通过创建一个来自于类型声明的构造类型（§ 20.4）而形成，它使用对应于类型参数的每一个类型实参。由于实例类型使用类型参数，因此只有在类型参数作用域内（类声明之内），它才是有效的。实例类型在类声明中是 `this` 的类型。对于非泛型类，实例类型就是声明类型。下面展示了几个类声明，以及它们的实例类型。

```
class A<T>           //实例类型: A<T>
{
    class B{}        //实例类型: A<T>.B
    class C<U>{}     //实例类型: A<T>.C<U>
}
class D{}           //实例类型: D
```

20.1.3 基类规范

在类声明中指定的基类可以是一个构造类型（§ 20.5）。基类其自身不能是一个类型参数，但在其作用域内可以包含类型参数。

```
class Extend<V>: V{} //错误，类型参数用做基类
```

泛型类声明不能使用 `System.Attribute` 作为直接或间接基类。

类声明中指定的基接口可以是构造接口类型（§ 20.5）。基接口自身不能是类型参数，但在其作用域内可以包含类型参数，下面的代码演示了如何实现和扩展构造类型。

```
class C<U,V>{}
interface I1<V>{}
class D:C<string , int>,I1<string>{}
class E<T>:C<int,T> ,I1<T>{}

```

泛型类型声明的基接口必须满足 § 20.3.1 中所描述的惟一性规则。

在一个类中的方法如果重写或实现了基类或接口的方法，那么它必须为特定类型提供合适的方法。下面的代码演示了方法如何被重写和实现。这将会在 § 20.1.10 中进一步解释。

```
class C<U,V>
{
    public virtual void M1(U x , List<V> y){...}
}
interface I1<V>
{
    V M2(V x);
}
class D:C<string , int>,I1<string>
{

```

```

        public override void M1(string x , List<int> y){...}
        public string M2(string x){...}
    }

```

20.1.4 泛型类的成员

泛型类的所有成员都可以使用封闭类（enclosing class）中的类型参数，不论是直接地使用还是作为构造类型的一部分。当在运行时使用特定的封闭构造类型时，类型参数的每次使用都由构造类型所提供的实际类型实参所代替。例如：

```

class C<V>
{
    public V f1;
    public C<V> f2=null;
    public C(V x){
        this.f1 = x;
        this.f2 = this;
    }
}
class Application
{
    static void Main(){
        C<int> x1= new C<int >(1);
        Console.WriteLine(x1.f1);           //打印 1
        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);           //打印 3.1415
    }
}

```

在实例函数成员之内，`this` 的类型就是声明的实例类型（§ 20.1.2）。

除了使用类型参数作为类型之外，在泛型类声明中的成员也遵循和非泛型类成员相同的规则。适用于特定种类成员的附加规则将在后面几节进行讨论。

20.1.5 泛型类中的静态字段

在一个泛型类声明中的静态变量，在相同封闭构造类型（§ 20.5.2）的所有实例中被共享，但在不同封闭构造类型的实例中，是不被共享的^{译注 2}。不管静态变量的类型包含那种类型参数，这些规则都适用。

例如：

```

class C<V>
{
    static int count = 0;
    public C(){
        count++;
    }
    public static int Count{
        get{return count;}
    }
}

```

^{译注 2} 这是很容易理解的，因为在运行时，不同的封闭构造类型，是属于不同的类型，比如 `List<int>` 和 `List<string>` 这二者的实例是不能共享静态变量的。

```

    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);    //打印 1
        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);    //打印 1
        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);    //打印 2
    }
}

```

20.1.6 泛型类中的静态构造函数

泛型类中的静态构造函数用于初始化静态字段，为每个从特定泛型类声明中创建的不同封闭构造类型，执行其他初始化。泛型类型声明的类型参数在作用域之内，可以在静态构造函数体内被使用。

如果下列情形之一发生，则一个新的封闭构造类类型将被首次初始化。

- 一个封闭构造类型的实例被创建；
- 封闭构造类型的任何静态成员被引用。

为了初始化一个新的封闭构造类类型，首先那个特定封闭类型的一组新静态字段（§ 20.1.5）将会被创建。每个静态字段都被初始化为其默认值（§ 5.2）。接着，将为这些静态字段而执行静态字段初始化器（§ 10.4.5.1）。最后将执行静态构造函数。

由于静态构造函数将为每个封闭构造类类型执行一次，因此对于无法在编译时通过约束（§ 20.7）进行的检查来说，在运行时去实施运行时检查将会很方便。例如，下面的类型使用静态构造函数检查一个类型参数是否是一个引用类型：

```

class Gen<T>
{
    static Gen(){
        if((object)T.default != null){
            throw new ArgumentException("T must be a reference type");
        }
    }
}

```

20.1.7 访问受保护的成员

在一个泛型类声明中，通过从泛型类构造的任何类型的实例，都可以对继承的受保护实例成员进行访问。尤其是，用于访问 § 3.5.3 中指定的 `protected` 和 `protected internal` 实例成员的规则，对于泛型使用如下的规则进行了扩充。

- 在一个泛型类 `G` 中，对于一个继承的受保护的实例成员 `M`，可以使用 `E.M` 形式的基本表达式对一个继承的受保护实例成员 `M` 进行访问，前提是 `E` 的类型是一

一个从 G 构造的类类型，或者一个继承于从 G 构造的类类型的类类型。

示例：

```
class C<T>
{
    protected T x;
}
class D<T> :C<T>
{
    static void F(){
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = T.default;
        di.x = 123;
        ds.x = "test";
    }
}
```

三个对 x 的赋值语句都是允许的，因为它们都通过从泛型构造的类类型的实例发生。

20.1.8 在泛型类中重载

在一个泛型类声明中的方法、构造函数、索引器和运算符可以被重载。但为了避免在构造类中出现歧义，这些重载是受约束的。在同一个泛型类声明中使用相同的名称声明的两个函数成员必须具有这样的参数类型，也就是封闭构造类型中不能出现两个成员使用相同的名称和签名。当考虑所有可能的封闭构造类型时，这条规则包含了在当前程序中目前不存在的类型实参，但它仍然是可能出现的^{译注 3}。在类型参数上的类型约束由于这条规则的目的而被忽略了。下面的例子根据这条规则展示了有效的和无效的重载：

```
interface I1<T> {...}
interface I2<T> {...}

class G1<U>
{
    long F1(U u);           //无效重载，G<int>将有使用相同签名的两个成员
    int F1(int i);
    void F2(U u1, U u2);    //有效重载，对于U没有类型参数
    void F2(int i, string s); //可能同时是int和string
    void F3(I1<U>a);        //有效重载
    void F3(I2<U>a);
    void F4(U a);           //有效重载
    void F4(U[] a);
}

class G2<U,V>
{
    void F5(U u, V v);      //无效重载，G2<int,int>将会有两个签名相同的成员
    void F5(V v, U u);
    void F6(U u, I1<V> v);  //无效重载，G2<I1<int>,int>将会有两个签名相同的成员
    void F6(I1<V> v, U u);
}
```

^{译注 3} 也就是在类型参数被替换成类型实参时，替换后的实参有可能导致出现两个成员使用相同的名字和签名。


```

        void F7(U u1, I1<V> v2);           //有效的重载, U不可能同时是V和I1<V>
        void F7(V v1, U u2);
        void F8(ref U u);                   //无效重载
        void F8(out V v);

        class C1{...}
        class C2{...}
        class G3<U, V> where U:C1 where V:C2
        {
            void F9(U u);                   //无效重载, 当检查重载时, 在U和V上的约束将被忽略
            void F9(V v);
        }

```

20.1.9 参数数组方法和类型参数

类型参数可以用在参数数组的类型中。例如, 给定声明:

```

class C<V>
{
    static void F(int x, int y, params V[] args);
}

```

方法的扩展形式的如下调用:

```

C<int>.F(10, 20);
C<object>.F(10, 20, 30, 40);
C<string>.F(10, 20, "hello", "goodbye");

```

对应于如下形式:

```

C<int>.F(10, 20, new int[]{});
C<object>.F(10, 20, new object[] {30, 40});
C<string>.F(10, 20, new string[] {"hello", "goodbye"});

```

20.1.10 重写和泛型类

和往常一样, 在泛型类中的函数成员可以重写基类中的函数成员。如果基类是一个非泛型类型或封闭构造类型, 那么任何重写函数成员不能有包含类型参数的组成类型。然而, 如果一个基类是一个开放构造类型, 那么重写函数成员可以使用其声明中的类型参数。当重写基类成员时, 基类成员必须通过替换类型实参来确定, 如 § 20.5.4 中所述。一旦基类的成员被确定, 则用于重写的规则和非泛型类是一样的。

下面的示例演示了对于现有的泛型其重写规则是如何工作的:

```

abstract class C<T>
{
    public virtual T F(){...}
    public virtual C<T> G(){...}
    public virtual void H(C<T> x){...}
}
class D:C<string>
{
    public override string F(){...}           //OK
    public override C<string> G(){...}         //OK
    public override void H(C<T> x){...}        //错误, 应该是C<string>
}

```

```

    }
    class E<T,U>:C<U>
    {
        public override U F(){...}           //OK
        public override C<U> G(){...}         //OK
        public override void H(C<T> x){...}    //错误, 应该是 C<U>
    }

```

20.1.11 泛型类中的运算符

泛型类声明可以定义运算符，它遵循与常规类相同的规则。类声明的实例类型 (§ 20.1.2) 必须以一种类似于运算符的常规规则的方式，在运算符声明中被使用：

- 一元运算符必须接受一个实例类型的单一参数。一元运算符“++”和“--”必须返回实例类型。
- 二元运算符的参数至少有一个必须是实例类型。
- 转换运算符的参数类型和返回类型都必须是实例类型。

下面在泛型类中展示了几个有效的运算符声明的例子：

```

class X<T>
{
    public static X<T> operator ++(X(T) operand){...}
    public static int operator *(X<T> op1, int op2){...}
    public static explicit operator X<T>(T value){...}
}

```

对于一个从源类型 *S* 到目标类型 *T* 的转换运算符，当应用 § 10.9.3 中的规则时，任何关联 *S* 或 *T* 的类型参数都被认为是惟一类型，它们与其他类型没有继承关系，并且在这些类型参数上的任何约束都将被忽略。

例如：

```

class C<T>{...}
class D<T>:C<T>
{
    public static implicit operator C<int>(D<T> value){...} //OK
    public static implicit operator C<T>(D<T> value){...}   //错误
}

```

第一个运算符声明是允许的，这是因为由于 § 10.9.3 的原因，*T* 和 *int* 被认为是没有关系的惟一类型。然而，第二个运算符是一个错误，因为 *C<T>* 是 *D<T>* 的基类。

给定前面的例子，为某些类型实参声明运算符、指定已经作为预定义转换而存在的转换是可能的。

```

struct Nullable<T>
{
    public static implicit operator Nullable<T>(T value){...}
    public static explicit operator T(Nullable<T> value){...}
}

```

当类型 *object* 作为 *T* 的类型实参被指定时，第二个运算符声明了一个已经存在的转换（从任何类型到 *object* 可以是隐式的，也可以是显式的转换）。

在两个类型之间存在预定义转换的情形下，在这些类型上的所有用户定义的转换都将被忽略。尤其是：

- 如果存在从类型 S 到类型 T 的预定义的隐式转换 (§ 6.1)，那么所有用户定义的转换（隐式的或显式的）都将被忽略。
- 如果存在从类型 S 到类型 T 的预定义的显式转换，那么所有用户定义的从类型 S 到类型 T 的显式转换都将被忽略。但用户定义的从 S 到 T 的隐式转换仍会被考虑。

对于除了 `object` 的所有类型，由 `Nullable<T>` 类型声明的运算符都不会与预定义的转换冲突。例如：

```
void F(int i , Nullable<int> n){
    i = n;                //错误
    i = (int)n;           //用户定义的显式转换
    n = i;                //用户定义的隐式转换
    n = (Nullable<int>)i;  //用户定义的隐式转换
}
```

然而，对于类型 `object`，预定义的转换在所有情况隐藏了用户定义转换，有一种情况除外：

```
void F(object o , Nullable<object> n){
    o = n;                //预定义装箱转换
    o = (object)n;         //预定义装箱转换
    n = o;                //用户定义隐式转换
    n = (Nullable<object>)o; //预定义取消装箱转换
}
```

20.1.12 泛型类中的嵌套类型

泛型类声明可以包含嵌套类型声明。封闭类的类型参数可以在嵌套类型中使用。嵌套类型声明可以包含附加的类型参数，它只适用于该嵌套类型。

包含在泛型类声明中的每个类型声明是隐式的泛型类型声明。当编写一个嵌套在泛型类型内的类型引用时，包含构造类型，包括它的类型实参，必须被命名。然而，在外部类中，内部类型可以被无限制地使用；当构造一个内部类型时，外部类的实例类型可以被隐式地使用。下面的例子展示了三个不同的方法，它们都引用从 `Inner` 创建的构造类型，并且它们都是正确的，其中前两个是等价的：

```
class Outer<T>
{
    class Inner<U>
    {
        static void F(T t , U u){...}
    }
    static void F(T t){
        Outer<T>.Inner<string>.F(t,"abc"); //这两个语句有同样的效果
        Inner<string>.F(t,"abc");
        Outer<int>.Inner<string>.F(3,"abc"); //这个类型是不同的
        Outer.Inner<string>.F(t , "abc"); //错误，Outer 需要类型参数
    }
}
```

尽管这不是一种很好的编程风格，但嵌套类型中的类型参数可以隐藏一个成员，或者外部类型中声明的一个类型参数。

```
class Outer<T>
{
    class Inner<T>    //有效，隐藏了 Outer 的 T
    {
        public T t;    //引用 Inner 的 T
    }
}
```

20.1.13 应用程序入口点

应用程序入口点方法（§ 3.1）不能存在于一个泛型类声明中。

20.2 泛型结构声明

与类声明一样，结构声明可以有可选的类型参数。

struct-declaration: (结构声明:)

```
attributesopt struct-modifiersopt struct identifier type-parameter-listopt struct-
interfacesopt type-parameter-constraints-clausesopt struct-bodyopt ;opt
(特性可选 结构修饰符可选 struct 标识符 类型参数列表可选 结构接口可选 类型参数
约束语句可选 结构体;可选)
```

除了 § 11.3 中为结构声明而指出的差别之外，泛型类声明的规则也适用于泛型结构声明。

20.3 泛型接口声明

接口也可以定义可选的类型参数

interface-declaration: (接口声明:)

```
attributesopt interface-modifiersopt interface identifier type-parameter-listopt
interface-baseopt type-parameter-constraints-clausesopt interface-bodyopt ;opt
(特性可选 接口修饰符可选 interface 标识符 类型参数列表可选 基接口可选 类型
参数约束语句可选 接口体;可选)
```

使用类型参数声明的接口是一个泛型接口声明。除了所指出的那些，泛型接口声明遵循和常规结构声明相同的规则。

在接口声明中的每个类型参数在接口的声明空间（§ 3.3）定义了一个名字。在一个接口上的类型参数的作用域（§ 3.7）包括基接口、类型约束语句和接口体。在其作用域之内，类型参数可以用做一个类型。应用到接口上的类型参数和应用到类（§ 20.1.1）上的类型参数具有相同的限制。

泛型接口中的方法与泛型类（§ 20.1.8）中的方法遵循相同的重载规则。

20.3.1 实现接口的惟一性

由泛型类型声明实现的接口必须为所有可能的构造类型保留惟一性。没有这条规则，将不可能为特定的构造类型确定正确的调用方法。例如，假定一个泛型类声明允许如下写法：

```
interface I<T>
{
    void F();
}
class X<U, V>:I<U>,I<V> //错误, I<U>和 I<V>冲突
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

如果允许这么写，那么下面的情形将无法确定执行那段代码：

```
I<int> x = new X<int ,int>();
x.F();
```

为了确定一个泛型类型声明的接口列表是有效的，可以按下面的步骤进行。

- 让 L 成为在泛型类、结构或接口声明 C 中指定的接口的列表。
- 将任何已经在 L 中的接口的基接口添加到 L。
- 从 L 中删除任何重复的接口。
- 在类型实参被替换到 L 后，如果任何从 C 创建的可能构造类型，导致在 L 中的两个接口是同一的，那么 C 的声明是无效的。当确定所有可能的构造类型时，对于约束声明不予考虑。

在类声明 X 之上，接口列表 L 由 I<U>和 I<V>组成。该声明是无效的，因为任何使用相同类型 U 和 V 的构造类型，都将导致这两个接口是同一的。

20.3.2 显式接口成员实现

使用构造接口类型的显式接口成员实现本质上与简单接口类型方式上是相同的。和以往一样，显式接口成员实现必须由一个指明哪个接口被实现的接口类型来限定。该类型可能是一个简单接口或构造接口，如下面的例子所示：

```
interface IList<T>
{
    T[] GetElements();
}
interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key , V value);
}
class List<T>:IList<T>,IDictionary<int , T>
{
    T[] IList<T>.GetElements() {...}
```

```

    T IDictionary<int, T>.this[int index]{...}
    void IDictionary<int, T>.Add(int index, T value){...}
}

```

20.4 泛型委托声明

委托声明可以包含类型参数。

delegate-declaration:

```

attributesopt delegate-modifiersopt delegate return-type identifier type-parameter-listopt
    (formal-parameter-listopt) type-parameter-constraints-clausesopt;
    (委托声明: 特性可选 委托修饰符可选 delegate 返回类型 标识符 类型参数列表可选
    (正式参数列表可选) 类型参数约束语句可选)

```

使用类型参数声明的委托是一个泛型委托声明。委托声明只有在支持类型参数列表时，才能支持类型参数约束语句 (§ 20.7)。除了所指出的之外，泛型委托声明和常规的委托声明遵循相同的规则。泛型委托声明中的每个类型参数在与委托关联的特定声明空间 (§ 3.3) 定义了一个名字。在委托声明中的类型参数的作用域包括返回类型、正式参数列表和类型参数约束语句。

像其他泛型类型声明一样，必须给定类型实参以形成构造委托类型。构造委托类型的参数和返回值，由委托声明中构造委托类型的每个类型参数对应的实参替代所形成。而结果返回类型和参数类型用于确定什么方法与构造委托类型兼容。例如：

```

delegate bool Predicate<T>(T value);
class X
{
    static bool F(int i){...}
    static bool G(string s){...}
    static void Main(){
        Predicate<int> p1 = F;
        Predicate<string> p2=G;
    }
}

```

注意上面的 Main 方法中的两个赋值等价于下面的较长形式：

```

static void Main(){
    Predicate<int> p1 = new Predicate<int>(F);
    Predicate<string> p2 = new Predicate<string>(G);
}

```

由于方法组转换，较短的形式也是可以的，这在 § 21.9 中有说明。

20.5 构造类型

泛型类型声明自身并不表示一个类型。相反，泛型类型声明通过应用类型实参的方式用做形成许多不同类型的“蓝图”。类型实参 (type argument) 被写在尖括号 (<>) 之间，并紧随泛型类型声明名称之后。使用至少一个实参而命名的类型称为**构造类型**

(**constructed type**)。构造类型可以用在语言中类型名称可以出现的大多数地方。

type-name: (类型名字:)

namespace-or-type-name (命名空间或类型名称)

namespace-or-type-name: (命名空间或类型名称:)

identifier type-argument-list_{opt} (标识符类型实参列表_{可选})

namespace-or-type-name.identifier (命名空间或类型名称.标识符)

type-argument-list_{opt} (类型实参列表_{可选})

构造类型也能用在表达式中作为简单名称 (§ 20.9.3), 当访问一个成员 (§ 20.9.4) 时也可以使用构造类型。

当一个命名空间或类型名称被计算时, 只有带有正确数量类型参数的泛型类型会被考虑。由此, 只要类型有不同数量的类型参数并且声明在不同的命名空间, 那么使用相同的标识符标识不同的类型是可能的。这对于在同一程序中混合使用泛型和非泛型类是很有用的。

```
namespace System.Collections
{
    class Queue(...)
}
namespace System.Collections.Generic
{
    class Queue<ElementType>(...)
}
namespace MyApplication
{
    using System.Collections;
    using System.Collections.Generic;
    class X
    {
        Queue q1;           //System.Collections.Queue
        Queue<int> q2;       //System.Collections.Generic.Queue
    }
}
```

在这些代码中对于名称查找的详细规则在 § 20.9 中进行了描述, 而对于模糊性的决议在 § 20.6.5 中进行了描述。

类型名称可能标识一个构造类型, 尽管它没有直接指定类型参数。这种情况在一个类型嵌套在一个泛型类声明中时就会出现, 并且包含声明的实例类型将因为名称查找 (§ 20.1.12) 而被隐式地使用。

```
class Outer<T>
{
    public class Inner(...)
    public Inner i;           //i 的类型是 Outer<T>.Inner
}
```

在不安全代码中, 构造类型不能用做非托管类型 (§ 18.2)。

20.5.1 类型实参

在一个类型实参列表中的每个实参都只是一个类型。

type-argument-list: (类型实参列表:)

<type-arguments> (<类型实参>)

type-arguments: (类型实参:)

type-argument (类型实参)

type-arguments, type-argument (类型实参, 类型实参)

type-argument: (类型实参:)

type (类型)

类型实参反过来也可以是构造类型或类型参数。在不安全代码 (§ 18) 中, 类型实参不能是指针类型。每个类型实参必须遵循对应类型参数 (§ 20.7.1) 上的任何约束。

20.5.2 开放类型和封闭类型

所有类型都可以被分为**开放类型** (open type) 或**封闭类型** (closed type)。开放类型是包含类型参数的类型。更明确的说法是:

- 类型参数定义了一个开放类型。
- 数组类型只有当其元素是开放类型时才是开放类型。
- 构造类型只有当其类型实参中的一个或多个是开放类型时才是开放类型。

非开放类型都是封闭类型。

在运行时, 在泛型类型声明中的所有代码都在一个封闭构造类型的上下文中执行, 这个封闭构造类型是通过将类型实参应用到泛型声明中创建的。在泛型类型中的每个类型实参被绑定到一个特定运行时类型。所有语句和表达式的运行时处理总是针对封闭类型发生, 而开放类型只在编译时处理的过程中发生。

每个封闭构造类型都有它自己的一组静态变量, 它们并不被其他封闭类型共享。因为在运行时不存在开放类型, 所以开放类型没有关联的静态变量。如果两个封闭构造类型是从同一个类型声明构造的, 并且对应的类型实参也是相同的类型, 那么它们就是相同的类型。

20.5.3 构造类型的基类和接口

构造类类型有一个直接基类, 很像是一个简单类类型。如果泛型类声明没有指定基类, 则其基类为 `object`。如果在泛型类声明中指定了基类, 那么对这个构造类型的基类, 可以通过把在基类声明中的每个类型参数替代为这个构造类型的对应类型实参而得到。给定泛型类声明:

```
class B<U , V>{...}
class G<T>:B<string , T[]>{...}
```

构造类型 `G<int>` 的基类将会是 `B<string , int[]>`。

类似地，构造类、结构和接口类型有一组显式的基接口。显式基接口通过接受泛型类型声明中的显式基接口声明和某种替代而形成，这种替代是把在基接口声明中的每个类型参数，替代为构造类型的对应类型实参。

一个类型的所有基类和基接口集合可以通过递归地得到基类和接口的直接基类与接口而形成。例如，给定泛型类声明：

```
class A {...}
class B<T>:A{...}
class C<T>:B<IComparable<T>>{...}
class D<T>:C<T[]>{...}
```

`D<int>` 的基类是 `C<int[]>`，`B<IComparable<int[]>>`，`A` 和 `object`。

20.5.4 构造类型的成员

构造类型的非继承成员通过将成员声明中的每个类型参数，替代为这个构造类型的对应类型实参而得到。

例如，给定泛型类声明。

```
class Gen<T, U>
{
    public T[,] a;
    public void G(int i , T t , Gen<U, T> gt){...}
    public U Prop{get{...} set{...}}
    public int H(double d){...}
}
```

构造类型 `Gen<int[],IComparable<string>>` 有如下的成员。

```
public int[,] a;
public void G(int i , int[] t , Gen<IComparable<string>,int[]> gt){...}
public IComparable<string> Prop {get{...} set{...}}
public int H(double d){...}
```

注意替代处理是基于类型声明的语义意义的，并不是简单的基于文本的替代。泛型类声明 `Gen` 中的成员 `a` 的类型是“`T` 的二维数组”，因此在前面实例化类型中的成员 `a` 的类型是“`int` 型的一维数组的二维数组”或 `int[,]`。

构造类型的继承成员以一种相似的方法获得。首先，直接基类的所有成员是已经确定的。如果基类自身是构造类型，这可能包括当前规则的递归应用。然后，每一个继承成员通过将成员声明中的每个类型参数替代为构造类型对应类型实参而被转换。

```
class B<U>
{
    public U F(long index){...}
}
class D<T>:B<T[]>
{
    ...
}
```

```

        public T G(string s){...}
    }

```

在上面的例子中，构造类型 $D<int>$ 的非继承成员 `public int G(string s)` 通过替代类型参数 T 的类型实参 `int` 而得到。 $D<int>$ 也有一个从类声明 B 继承而来的成员。这个继承成员通过首先确定构造类型 $B<T[]>$ 的成员而被确定， $B<T[]>$ 成员的确定是通过将 U 替换为替换为 $T[]$ ，产生 `public T[] F(long index)`。然后，类型实参 `int` 替换了类型参数 T ，产生继承成员 `public int[] F(long index)`。

20.5.5 构造类型的可访问性

如果构造类型 $C<T_1, \dots, T_N>$ 的所有部分 C, T_1, \dots, T_N 可访问，那么它就是可访问的。例如，如果泛型类型名 C 是 `public`，并且所有类型参数 T_1, \dots, T_N 也是 `public`，那么构造类型的可访问性也是 `public`。如果类型名或类型实参之一是 `private`，那么构造类型的可访问性是 `private`。如果类型实参之一可访问性是 `protected`，另一个是 `internal`，那么构造类型的可访问性仅限于该类及本程序集之内的子类。

更准确地说，构造类型的可访问域是它各个组成要素的可访问域的交集。由此可见，假设一个方法有一个返回类型，或者实参类型；如果它是构造类型，而在这个类型有一个组成要素部分是 `private`，那么，这个方法具有 `private` 形式的可访问域（参见 § 3.5）。

20.5.6 转换

构造类型遵循与非泛型类型相同的转换规则（§ 6）。当应用这些规则时，构造类型的基类和接口必须按 § 20.5.3 中所描述的方式确定。

除了那些在 § 6 中所描述的之外，构造引用类型之间不存在特别的转换。尤其是，与数组类型不同，构造引用类型不允许“`co-variant`”^{译注 4}转换。也就是说，类型 $List$ 不能转换到类型 $List<A>$ （无论是隐式或显式），即使是 B 派生于 A 也是如此。同样，也不存在从 $List$ 到 $List<object>$ 的转换。

关于这一点的基本原理是很简单的：如果可以转换到 $List<A>$ ，很显然你可以将一个类型 A 的值存储到这个 `list` 中。这将破坏 $List$ 类型列表中的每个对象总是类型 B 的值这种不变性，否则当在集合类上赋值时，将产生不可预料的错误。

转换的行为和运行时类型检查演示如下：

```

class A {...}
class B:A {...}
class Collection {...}
class List<T>:Collection {...}
class Test
{
    void F() {
        List<A> listA = new List<A>();
        List<B> listB = new List<B>();
    }
}

```

^{译注 4} `co-variant` 共变类型。

```

        Collection c1 = listA;           //OK, List<A>是一个集合
        Collection c2 = listB;           //OK, List<B>是一个集合
        List<A> a1 = listB;              //错误, 没有隐式的转换
        List<A> a2 = (List<A>)listB;     //错误, 没有显式的转换
    }
}

```

20.5.7 System.Nullable<T>类型

在.NET 基类库中定义了 `System.Nullable<T>` 泛型结构类型, 它表示一个可以为 `null` 的类型 `T` 的值。`System.Nullable<T>` 类型在很多情形下是很有用的, 例如可用于指示数据库表的可空列, 或者 XML 元素中的可选特性。

可以从 `null` 类型向任何由 `System.Nullable<T>` 类型构造的类型做隐式的转换。这种转换的结果就是 `System.Nullable<T>` 的默认值。也就是说, 可以这样写:

```

Nullable<int> x = null;
Nullable<string> y = null;

```

和下面的写法相同:

```

Nullable<int> x = Nullable<int>.default;
Nullable<string> y = Nullable<string>.default;

```

20.5.8 使用别名指令

使用别名可以命名一个封闭构造类型, 但不能命名一个没有提供类型实参的泛型类型声明。例如:

```

namespace N1
{
    class A<T>
    {
        class B{}
    }
    class C{}
}
namespace N2
{
    using W = N1.A;           //错误, 不能命名泛型类型
    using X = N1.A.B;         //错误, 不能命名泛型类型
    using Y = N1.A<int>;      //Ok, 可以命名封闭构造类型
    using Z = N1.C;           //Ok
}

```

20.5.9 特性

开放类型不能在特性内的任何地方使用。封闭构造类型可以用做特性的实参, 但不能用做特性名, 因为 `System.Attribute` 不可能是泛型类声明的基类型。

```
class A:Attribute
{
    public A(Type t){...}
}
class B<T>: Attribute{}           //错误, 不能将 Attribute 用做基类
class List<T>
{
    [A(typeof(T))] T t;           //错误, 在特性中有开放类型
}
class X
{
    [A(typeof(List<int>))] int x;   //OK, 封闭构造类型
    [B<int>] int y;                 //错误, 无效的特性名称
}
```

20.6 泛型方法

泛型方法是与特定类型相关的方法。除了常规参数之外，泛型方法还命名了在使用方法时需要提供的一组类型参数。泛型方法可以在类、结构或接口声明中声明，而它们本身可以是泛型或者非泛型的。如果一个泛型方法在一个泛型类型声明中被声明，那么方法体可以引用方法的类型参数和包含声明的类型参数。

class-member-declaration: (类成员声明:)

...

generic-method-declaration (泛型方法声明)

struct-member-declaration: (结构成员声明:)

...

generic-method-declaration (泛型方法声明)

interface-member-declaration: (接口成员声明:)

...

interface-generic-method-declaration (接口泛型方法声明)

泛型方法可以通过在方法的名字之后放置类型参数列表来声明。

generic-method-declaration: (泛型方法声明:)

generic-method-header **method-body** (泛型方法头 方法体)

generic-method-header: (泛型方法头:)

attributes opt **method-modifiers** opt **return-type** **member-name** **type-parameter-list**
(**formal-parameter-list** opt) **type-parameter-constraints-clauses** opt
(特性_{可选} 方法修饰符_{可选} 返回类型 成员名 类型参数列表 (正式参数列表_{可选})
类型参数约束语句_{可选})

interface-generic-method-declaration: (接口泛型方法声明:)

attributes opt **new** opt **return-type** **identifier** **type-parameter-list**
(**formal-parameter-list** opt) **type-parameter-constraints-clauses** opt ;
(特性_{可选} **new** _{可选} 返回类型 标识符 类型参数列表 (正式参数列表_{可选}) 类
类型参数约束语句_{可选} ;

类型参数列表和类型参数约束语句与泛型类型声明具有同样的语法和功能。由类型参数列表声明的类型参数作用域贯穿整个泛型方法声明，它可以用于形成贯穿包括返回值、方法体和类型参数约束语句（但不包括特性）在内的该作用域的类型。

方法的类型参数的名称与同一方法的常规参数名称不能相同。

如果满足给定 `test` 委托存在，则下面的例子将找到数组中的第一个元素。泛型委托在 § 20.4 中进行描述。

```
public delegate bool Test<T>(T item);
public class Finder
{
    public static T Find<T>(T[] items , Test<T> test){
        foreach(T item in items){
            if(test(item)) return item;
        }
        throw new InvalidOperationException("Item not found");
    }
}
```

泛型方法不能被声明为 `extern`。所有其他修饰符在泛型方法上都是有效的。

20.6.1 泛型方法签名

为了签名的比较，任何参数约束和类型参数的名字都将被忽略^{译注 5}，但类型参数的个数和从左到右的顺序位置却是很重要的。下面的例子展示了这条规则影响下的方法签名：

```
class A{}
class B {}
interface IX
{
    T F1<T>(T[] a , int i);    //错误，因为返回类型和类型参数名字无关紧要，
    void F1<U>(U[] a ,int i); //这两个声明都有相同的签名
    void F2<T>(int x);        //OK, 类型参数的数量是签名的一部分
    void F2(int x);
    void F3<T>(T t) where T: A; // 错误，约束不在签名考虑之列
    void F3<T>(T t) where T:B;;
}
```

泛型方法的重载采用一条与在泛型类型声明（§ 20.1.8）中管理方法重载相似的规则，进一步地进行了限制。两个使用相同名称和相同数量的类型实参的两个泛型方法声明，只能具有无封闭类型实参列表的参数类型，当它们以相同顺序应用到两个方法上时，将产生两个具有相同签名的方法。由于这条规则约束将不予考虑。例如：

```
class X<T>
{
    void F<U>(T t , U u){...}    //错误，X<int>.F<int> 产生了具有相同签名的两个方法
    void F<U>(U u, T t){...}    //
}
```

^{译注 5} 类型参数的名字只是一个占位符而已，比如类型 `F<U>` 和 `F<V>`，中的 `U, V` 都只是一个占位符，它们需要用实际的类型替代，因此，不能当成不同签名的依据。

20.6.2 虚拟泛型方法

泛型方法可以用 `abstract`, `virtual` 和 `override` 修饰符来声明。当为重载或接口实现进行方法匹配时, 将使用 § 20.6.1 中描述的签名匹配规则。当泛型方法重写在基类中声明的泛型方法, 或者实现基接口中的方法时, 为每个方法类型参数给定的约束在两个声明中必须是相同的, 在这里方法类型参数由原始位置按从左到右的顺序来标识。

```
abstract class Base
{
    public abstract T F<T,U>(T t, U u);
    public abstract T G<T>(T t) where T: IComparable;
}
class Derived:Base
{
    public override X F<X,Y>(X x ,Y y){...} //OK
    public override T G<T>(T t){...} //错误
}
```

F 的重写是正确的, 因为类型参数名称可以不同。G 的重写是错误的, 因为给定的类型参数约束 (在这里没有约束) 与被重写的方法不匹配。

20.6.3 调用泛型方法

泛型方法调用可以显式地指定类型实参列表, 或者省略类型实参列表而依靠类型推断来确定类型实参。关于方法调用的确切编译时处理 (包括泛型方法调用), 在 § 20.9.5 中进行了描述。当泛型方法不使用类型参数列表调用时, 类型推断将按 § 20.6.4 中所描述的方式进行。

下面的示例展示了在类型推断和类型实参替代参数列表后, 重载决策是如何发生的。

```
class Test
{
    static void F<T>(int x , T y){
        Console.WriteLine("One");
    }
    static void F<T>(T x , long y){
        Console.WriteLine("two");
    }
    static void Main(){
        F<int>(5,324); //ok, 打印 "one"
        F<byte>(5,324); //ok, 打印 "two"
        F<double>(5,324); //错误, 不明确
        F(5,324); //ok, 打印 "one"
        F(5,324L); //错误, 不明确
    }
}
```

20.6.4 类型实参推断

当不指定类型实参而调用泛型方法时, **类型推断 (type inference)** 处理将试图为这个

调用推断类型实参。类型推断的存在可以使调用泛型方法时，采用更方便的语法，并且可以避免程序员指定冗余的类型信息。例如，给定方法声明：

```
class Util
{
    static Random rand = new Random();
    static public T Choose<T>(T first , T second)
    {
        return (rand.Next(2) == 0)?first:second;
    }
}
```

也可以不显式指定类型实参而调用 `Choose` 方法：

```
int i = Util.Choose(5,213);           //调用 Choose<int>
string s = Util.Choose("foo","bar");  //调用 Choose<string>
```

通过类型推断，类型实参 `int` 和 `string` 将由方法的实参确定。

类型推断作为方法调用（§ 20.9.5）编译时处理的一部分发生，并且在调用的重载决策步骤之前发生。当在一个方法调用中指定特定的方法组时，类型实参不会作为方法调用的一部分而指定，类型推断将被应用到方法组中的每个泛型方法。如果类型推断成功，则被推断的类型实参被用于确定后续重载决策的实参类型。如果重载决策选择了将要调用的泛型方法，则被推断的类型实参将用做调用的实际类型实参。如果特定方法的类型推断失败，则这个方法将不参与重载决策。类型推断失败自身不会产生编译时错误。但当重载决策没能找到合适的方法时，将导致编译时错误。

如果所提供的实参个数与方法的参数个数不同，推断将立刻失败。否则，类型推断将为提供给方法的每个正式实参独立地进行处理。假定这个实参的类型为 `A`，对应参数的类型为 `P`，类型推断将按下列步骤关联类型 `A` 和 `P` 而发生。

- 如果以下任何一条成立，将不能从实参推断任何东西（但类型推断是成功的）
 - `P` 不调用方法的任何方法类型参数^{译注 6}
 - 实参是 `null` 字符
 - 实参是一个匿名方法
 - 实参是一个方法组
- 如果 `P` 是数组类型，`A` 是同秩（rank）的数组类型，那么使用 `A` 和 `P` 的元素类型相应地替换 `A` 和 `P`，并重复这个步骤。
- 如果 `P` 是数组类型，而 `A` 是不同秩的数组类型，那么泛型方法的类型推断失败。
- 如果 `P` 是方法的类型参数，那么对于这个实参的类型推断成功，并且 `A` 是那个类型实参所推断的类型。
- 否则，`P` 必须是构造类型。如果对于出现在 `P` 中的每个方法类型参数 M_x ，恰好可以确定一个类型 T_x ，使用每个 T_x 替换每个 M_x ，这将产生一个类型，对于这个类型，`A` 可以通过标准的隐式转换而被转换，那么对于这个实参的类型推断成功，对于每个 M_x ， T_x 就是推断的类型。方法类型参数约束（如果有的话），因为类型推断的原因将会被忽略。如果对于一个给定的 M_x ，没有 T_x 存在或者多

^{译注 6} 也就是 `P` 并不是方法的类型参数列表所列类型之一。

于一个 T_x 存在，那么泛型方法的类型推断将会失败（多于一个 T_x 存在的情形只可能是在 P 是一个泛型接口类型，并且 A 实现了该接口的多个构造版本时发生）。

如果所有的方法实参都通过先前的算法进行了成功的处理，那么由实参而产生的所有推断都将被汇聚。这组推断必须有如下的属性：

- 方法的每个类型参数必须有一个为其推断的类型实参。简而言之，这组推断必须是**完整的**（**complete**）；
- 如果类型参数出现多于一次，那么对那个类型参数的所做的推断都必须推断相同的类型实参。简而言之，这组接口必须是一**致的**（**consistent**）。

如果能够找到一组完整而一致的推断类型实参，那么对于一个给定的泛型方法和实参列表，类型推断就可以说是成功的。

如果泛型方法使用参数数组（§ 10.5.1.4）声明，那么类型推断将首先针对方法以其通常的方式执行。如果类型推断成功，结果方法是可用的，那么方法将以其通常形式参与重载决策。否则，类型推断将针对方法的扩展形式（§ 7.4.2.1）执行。

20.6.5 语法歧义

在 § 20.9.3 和 § 20.9.4 中的简单名称和成员访问对于表达式来说容易引起语法歧义。例如，语句：

```
F(G<A, B>(7));
```

可以被解释为对带有两个参数（ $G<A$ ）和（ $B>(7)$ ）的 F 的调用^{注 7}。同样，它还能被解释为对带有一个实参的 F 方法的调用，这个实参是对带有两个类型实参和一个正式实参的泛型方法 G 的调用。

如果表达式可以以两种不同的有效方法来解析，其中“>”可以被解析为运算符的全部或一部分，或者作为类型实参列表的一部分，那么来解析紧随“>”之后的标记将会被检查。如果它是如下之一：

```
( ) ] > : ; , . ?
```

那么“>”被解析为类型实参列表。否则“>”被解析为一个运算符。

20.6.6 对委托使用泛型方法

委托的实例可通过引用一个泛型方法声明而创建。委托创建表达式确切的编译时处理，包括引用泛型方法的委托创建表达式，在 § 20.9.6 中进行了描述。

当通过委托调用泛型方法时，所使用的类型实参将在委托实例化时确定。类型实参可以通过类型实参列表显式给定，或者通过类型推断（§ 20.6.4）确定。如果采用类型推断，委托的参数类型将用做推断处理过程的实参类型。委托的返回类型不用于推断。下面的例

^{注 7} 这种情况下“>”被解释为大于运算符。

子展示了为委托实例化表达式提供类型实参的方法。

```

delegate int D(string s, int i);
delegate int E();
class X
{
    public static T F<T>(string s, T t) {...}
    public static T G<T>() {...}
    static void Main() {
        D d1 = new D(F<int>);           //ok, 类型实参被显式给定
        D d2 = new D(F);                 //ok, int 作为类型实参被推断
        E e1 = new E(G<int>);           //ok, 类型实参被显式给定
        E e2 = new E(G);                 //错误, 不能从返回类型推断
    }
}

```

在上面的例子中，非泛型委托类型使用泛型方法来实例化。也可以使用泛型方法创建一个构造委托类型（§ 20.4）的实例。在所有情形下，当委托实例被创建时，都给定或推断类型实参，当委托被调用时，可以不用提供类型实参列表（§ 15.3）。

20.6.7 非泛型属性、事件、索引器或运算符

属性、事件、索引器和运算符自身可以没有类型参数（尽管它们可以出现在泛型类中，并且可从一个封闭类中使用类型参数）。如果需要一个类似属性的泛型构件，那么必须使用泛型方法。

20.7 约束

泛型类型和方法声明可以通过在声明中包含类型参数约束语句，有选择地指定类型参数约束。

type-parameter-constraints-clauses:（类型参数约束语句:）

type-parameter-constraints-clause（类型参数约束语句）

type-parameter-constraints-clauses type-parameter-constraints-clause（类型参数约束语句类型参数约束语句）

type-parameter-constraints-clause:（类型参数约束语句:）

where type-parameter :type-parameter-constraints（**where** 类型参数: 类型参数约束）

type-parameter-constraints:（类型参数约束:）

class-constraint（类约束）

interface-constraints（接口约束）

constructor-constraint（构造函数约束）

class-constraint, interface-constraints（类约束, 接口约束）

class-constraint, constructor-constraint（类约束, 构造函数约束）

interface-constraints, constructor-constraint（接口约束, 构造函数约束）

class-constraint, interface-constraints, constructor-constraint（类约束, 接口约束,

构造函数约束)

class-constraint: (类约束:)

 class-type (类类型)

interface-constraints: (接口约束:)

 interface-constraint (接口约束)

 interface-constraints , interface-constraint (接口约束, 接口约束)

interface-constraint: (接口约束:)

interface-type (接口类型)

constructor-constraint: (构造函数约束:)

 new ()

每个类型参数约束语句由标记 **where** , 后面紧跟类型参数的名字, 然后紧跟着冒号和类型参数的约束列表组成。对每个类型参数只能有一个 **where** 从句, 但 **where** 从句可以以任何顺序列出。与属性访问器中的 **get** 和 **set** 标记相似, **where** 标记不是关键字。

Where 从句中给定的约束列表可以以下列顺序包含下列组件: 一个单一的类约束、一个或多个接口约束, 以及构造函数约束 **new ()**。

如果约束是类类型或者接口类型, 则这个类型指定类型参数必须支持的每个类型实参的最小“基类型”。无论什么时候使用一个构造类型或者泛型方法, 在编译时类型实参对于类型参数上的约束都会被检查。所提供的类型实参必须派生自或者实现为了该类型参数而给定的所有约束。

被指定为类约束的类型必须遵循下面的规则。

- 该类型必须是类类型。
- 该类型必须不是密封的。
- 该类型不能是如下的类型: **System.Array**, **System.Delegate**, **System.Enum** 或者 **System.ValueType**。
- 该类型不能是 **object**。由于所有类型派生自 **object**, 因此即便容许的话这种约束也不会有什么作用。
- 对于给定类型参数的约束最多可以是一个类类型。

作为接口约束而被指定的类型必须满足如下的规则。

- 该类型必须是接口类型。
- 在一个给定的 **where** 语句中, 相同的类型不能被指定多次。

在很多情况下, 约束可以包含任何关联类型的类型参数或者方法声明作为构造类型的一部分, 并且可以包括正在被声明的类型, 但约束不能是单一的类型参数。

被指定为类型参数约束的任何类或者接口类型, 必须至少与泛型类型或正在被声明的方法具有相同的可访问性 (§ 10.5.4)。

如果一个类型参数的 **where** 语句包括 **new()**形式的构造函数约束, 则可以使用 **new** 运算符创建该类型 (§ 20.8.2) 的实例是可能的。用于带有一个构造函数约束的类型参数的任何类型实参必须有一个无参的构造函数 (详细情形参看 § 20.7.1)。

下面是可能的约束的示例

```
interface IPrintable
```

```

    {
        void Print();
    }
    interface IComparable<T>
    {
        int CompareTo(T value);
    }
    interface IKeyProvider<T>
    {
        T GetKey();
    }
    class Printer<T> where T:IPrintable{...}
    class SortedList<T> where T: IComparable<T>{...}
    class Dictionary<K,V>
        where K:IComparable<K>
        where V: IPrintable,IKeyProvider<K>,new()
    {
        ...
    }

```

下面的例子是一个错误，因为它试图直接使用一个类型参数作为约束：

```
class Extend<T , U> where U:T{...}    //错误
```

被约束的类型参数类型的值可以用于访问约束暗示的实例成员。对于如下示例：

```

interface IPrintable
{
    void Print();
}
class Printer<T> where T:IPrintable
{
    void PrintOne(T x)
    {
        x.Pint();
    }
}

```

其中，IPrintable 的方法可以在 x 上被直接调用，因为 T 被约束为总是实现 IPrintable。

20.7.1 满足约束

无论什么时候使用构造类型或者引用泛型方法，所提供的类型实参都将针对声明在泛型类型或者方法中的类型参数约束做出检查。对于每个 where 语句，对应于命名的类型参数的类型实参 A 将按如下内容针对每个约束做出检查。

- 如果约束是一个类类型或者接口类型，让 C 表示提供的类型实参的约束，该类型实参将替代出现在约束中的任何类型参数。为了满足约束，类型 A 必须是按如下方式转化为类型 C 的：
 - 同一转换 (§ 6.1.1)
 - 隐式引用转换 (§ 6.1.4)
 - 装箱转换 (§ 6.1.5)
 - 从类型参数 A 到类型参数 C 的隐式转换 (§20.7.4)。

- 如果约束是 `new()`，则类型实参 `A` 不能是 `abstract`，并且必须有一个公有的无参的构造函数。如果如下之一是真，则这将可以得到满足：
 - `A` 是一个值类型（如 § 4.1.2 中所描述的，所有值类型都有一个公有默认构造函数）。
 - `A` 是一个非 `abstract` 类，并且 `A` 包含一个无参公有构造函数。
 - `A` 是一个非 `abstract` 类，并且有一个默认构造函数（§ 10.10.4）。

如果给定的类型实参不能满足一个或多个类型参数的约束，将会出现编译时错误。

因为类型参数是不被继承的，所以约束也从不被继承。在下面的例子中，`D` 必须在其类型参数 `T` 上指定约束，以满足由基类 `B<T>` 所施加的约束。相反，类 `E` 不需要指定约束，因为对于任何 `T`，`List<T>` 实现了 `IEnumerable` 接口。

```
class B<T> where T: IEnumerable{...}
class D<T>:B<T> where T:IEnumerable{...}
class E<T>:B<List<T>>{...}
```

20.7.2 类型参数上的成员查找

在由类型参数 `T` 给定的类型中，成员查找的结果取决于为 `T` 所指定的约束（如果有的话）。如果 `T` 没有约束或者只有 `new()` 约束，则在 `T` 上的成员查找，与在 `object` 上的成员查找一样，返回一组相同的成员。否则，成员查找的第一个阶段，将考虑 `T` 所约束的每个类型的所有成员，结果将会被合并，然后将隐藏成员从合并结果中删除。

在泛型出现之前，成员查找总是或者返回一组在类中惟一声明的成员，或者返回一组在接口中惟一声明的成员，并且类型可能是 `object`。在类型参数上的成员查找做出了一些改变。当一个类型参数有一个类约束和一个或多个接口约束时，成员查找可以返回一组成员，这些成员有一些是在类中声明的，还有一些是在接口中声明的。下面的附加规则处理了这种情况。

- 在成员查找（§ 20.9.2）过程中，在除了 `object` 之外的类中声明的成员隐藏了在接口中声明的成员。
- 在方法（§ 7.5.5.1）和索引器（§ 7.5.6.2）的重载决策过程中，如果任何可用成员在一个不同于 `object` 的类中声明，那么在接口中声明的所有成员都将从被考虑的成员集合中删除。

这些规则只有在将一个类约束和接口约束绑定到类型参数上时才有效。通俗一点说就是，在类约束中定义的成员，对于在接口约束中的成员来说总是首选。

20.7.3 类型参数和装箱

当一个结构类型重写继承于 `System.Object` (`Equals`, `GetHashCode` 或 `ToString`) 的虚拟方法时，通过结构类型的实例调用虚拟方法将不会导致装箱。即使结构用做一个类型参数，并且通过类型参数类型的实例而发生调用，情况也是如此。例如：

```
using System;
```

```

struct Counter
{
    int value;
    public override string ToString()
    {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T:new(){
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }

    static void Main()
    {
        Test<Counter>();
    }
}

```

程序的输出如下：

```

1
2
3

```

尽管推荐不要让 `ToString` 带有附加效果（side effect）^{注8}，但这个例子说明了对于三次 `x.ToString()` 的调用不会发生装箱。

当在一个约束的类型参数上访问一个成员时，装箱绝不会隐式地发生。例如，假定一个接口 `ICounter` 包含了一个方法 `Increment`，该方法可以用来修改一个值。如果 `ICounter` 用做一个约束，则 `Increment` 方法的实现将通过 `Increment` 在其上被调用的变量的引用而被调用，这个变量不是一个装箱拷贝。

```

using System;
interface ICounter
{
    void Increment();
}
struct Counter:ICounter
{
    int value;
    public override string ToString(){
        return value.ToString();
    }
    void ICounter.Increment(){
        value++;
    }
}

class Program
{

```

^{注8} 在程序中重写 `ToString` 时，一般不推荐添加这种类似的计算逻辑，因为它的这种结果变化不易控制，增加了调试程序的复杂性。


```

static void Test<T>() where T:new (), ICounter{
    T x = new T();
    Console.WriteLine(x);
    x.Increment();           //修改 x
    Console.WriteLine(x);
    ((ICounter)x).Increment(); //修改 x 的装箱拷贝
    Console.WriteLine(x);
}
static void Main(){
    Test<Counter>();
}

```

对 Increment 的首次调用修改了变量 *x* 的值。这与第二次调用 Increment 是不等价的，第二次修改的是 *x* 装箱后的拷贝，因此程序的输出如下：

```

0
1
1

```

20.7.4 包含类型参数的转换

在类型参数 *T* 上允许的转换，取决于为 *T* 所指定的约束。所有约束的或非约束的类型参数，都可以有如下转换。

- 从 *T* 到 *T* 的隐式的同一转换。
- 从 *T* 到 *object* 的隐式转换。在运行时，如果 *T* 是一个值类型，则将作为一个装箱转换进行。否则，它将作为一个隐式的引用转换来执行。
- 从 *object* 到 *T* 的隐式转换。在运行时，如果 *T* 是一个值类型，则将作为一个取消装箱操作执行。否则它将作为一个显式的引用转换来执行。
- 从 *T* 到任何接口类型的显式转换。在运行时，如果 *T* 是一个值类型，则将作为一个装箱转换执行。否则，它将作为一个显式的引用转换执行。
- 从任何接口类型到 *T* 的隐式转换。在运行时，如果 *T* 是一个值类型，则将作为一个取消装箱操作执行。否则，它将作为一个显式的引用转换执行。

如果类型参数 *T* 指定一个接口 *I* 作为约束，将存在下面的附加转换。

- 从 *T* 到 *I* 的隐式转换，以及从 *T* 到 *I* 的任何基接口类型的转换。在运行时，如果 *T* 是一个值类型，则将作为一个装箱转换而进行。否则，它将作为一个隐式地引用转换而进行。

如果类型参数 *T* 指定类型 *C* 作为约束，将存在下面的附加转换：

- 从 *T* 到 *C* 的隐式引用转换，从 *T* 到任何 *C* 从中派生的类，以及从 *T* 到任何 *C* 实现的接口。
- 从 *C* 到 *T* 的显式引用转换，从 *C* 从中派生的类^{16.19}到 *T*，以及 *C* 实现的任何接口到 *T*。
- 如果存在从 *C* 到 *A* 的隐式用户定义转换，从 *T* 到 *A* 的隐式用户定义转换。

^{16.19} *C* 的基类或其基类的基类等。

- 如果存在从 A 到 C 的显式用户定义转换，从 A 到 T 的显式用户定义转换。
- 从 null 类型到 T 的隐式引用转换

一个带有元素类型 T 的数组类型具有 object 和 System.Array 之间的相互转换 (§ 6.1.4, § 6.2.3)。如果 T 有作为约束而指定的类类型 C，将有如下附加规则。

- 从带有元素类型 T 的数组类型 A_T 到带有元素类型 U 的数组类型 A_U 的隐式引用转换存在，并且如果下列二者成立的话，将存在从 A_U 到 A_T 显式引用转换。
 - A_T 和 A_U 有相同数量的维数。
 - U 是下列之一：C，C 从中派生的类型，C 所实现的接口，作为在 T 上的约束而指定的接口 I，或 I 的基接口。

上面的规则不允许从非约束类型参数到非接口类型的直接隐式转换，这可能有点奇怪。其原因是为了防止混淆，并且使得这种转换的语义更明确。例如，考虑下面的声明：

```
class X<T>
{
    public static long F(T t){
        return (long)t;    // 错误，不允许显式转换
    }
}
```

如果 t 到 int 的直接显式转换是允许的，很容易让人以为 X<int>.F(7) 将返回 7L。但实际上不是，因为标准的数值转换只有当类型在编译时是已知的时候才被考虑。为了使语义更清楚，上面的例子必须按如下形式编写。

```
class X<T>
{
    public static long F(T t)
    {
        return (long)(object)t;    //ok; 允许转换
    }
}
```

20.8 表达式和语句

某些表达式和语句的操作针对泛型进行了修改。这一节将介绍这些改变。

20.8.1 默认值表达式

默认值表达式用于获得一个类型的默认值 (§ 5.2)。通常一个默认值表达式用于类型参数，因为如果类型参数是一个值类型或引用类型，它可能不是已知的（不存在从 null 类型到类型参数的转换）。

primary-no-array-creation-expression: (基本无数组创建表达式:)

...

default-value-expression (默认值表达式)

default-value-expression: (默认值表达式:)

primary-expression.default (基本表达式.default)

predefined-type.default (预定义类型.default)

如果在一个默认值表达式中使用一个基本表达式, 并且这个基本表达式不是一个类型, 那么将会导致编译时错误。然而在 § 7.5.4.1 中描述的规则也适用于形成 E.default 的构件。

如果默认值表达式的左边针对一个引用类型在运行时被计算, 则结果是将 null 转换到那个类型。如果默认值表达式的左边针对一个值类型在运行时被计算, 则结果是值类型的默认值 (§ 4.1.2)。

如果类型是一个具有类约束的引用类型或类型参数, 则默认值表达式是一个常量表达式 (§ 7.15)。此外, 如果类型是下列值之一, 则默认值表达式是一个常量表达式: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal 或 bool。

20.8.2 对象创建表达式

对象常见表达式的类型可以是一个类型参数。当类型参数被作为对象创建表达式中的类型而指定时, 下面两个条件必须都具备, 否则将会出现编译时错误:

- 实参列表必须删除。
- 必须为类型参数指定 new () 形式的构造函数约束。

通过创建一个类型参数被绑定到的运行时类型的实例, 并调用该类型的默认构造函数, 就可以执行对象创建表达式。运行时类型可以是引用类型或者值类型。

20.8.3 typeof 运算符

typeof 运算符可以用于类型参数。其结果是被绑定到类型参数的运行时类型的 System.Type 对象。typeof 运算符也可以被用于构造类型。

```
class X <T>
{
    public static void PrintTypes(){
        Console.WriteLine(typeof(T).FullName);
        Console.WriteLine(typeof(X<X<T>>).FullName);
    }
}
class M
{
    static void Main()
    {
        X<int>.PrintTypes();
    }
}
```

上面的程序将打印如下:

```
System.Int32
X<X<System.Int32>>
```

typeof 运算符不能用于没有指定类型实参的泛型类型声明的名字。

```

class X<T>{...}
class M
{
    static void Main()
    {
        Type t = typeof(X);           //错误, x 需要类型实参
    }
}

```

20.8.4 引用相等运算符

如果 T 由一个类约束而约束, 引用类型相等运算符 (§ 7.9.6) 可以用于比较类型参数 T 的值。

引用类型相等运算符的用处是可以让类型参数 T 的实参很容易地与其他为 `null` 的实参进行比较, 即使 T 没有类约束也是如此。在运行时, 如果 T 是一个值类型, 比较的结果将是 `false`。

下面的例子检查一个非约束类型参数类型的实参是否是 `null`。

```

class C<T>
{
    void F(T x)
    {
        if(x==null) throw new ArgumentNullException();
        ...
    }
}

```

即使 T 可以表示一个值类型, `x==null` 构件也是允许的, 并且当 T 是值类型时, 其结果被简单地定义为 `false`。

20.8.5 is 运算符

在开放类型上的 `is` 运算符操作遵循通常的规则 (§ 7.9.9)。如果 e 或 T 的编译时类型是一个开放类型, 那么在运行时对于 e 和 T 将总是执行动态类型检查。

20.8.6 as 运算符

只有在 T 有一个类约束时, 类型参数 T 才可以用在 `as` 运算符的右边。这种限制是需要的, 因为值 `null` 可能作为运算符的结果返回。

```

class X
{
    public T F<T>(object o) where T:Attribute
    {
        return o as T;           //ok, T 有一个类约束
    }
    public T G<T>(object o)
    {
        return o as T;           //错误, T 没有约束
    }
}

```

在 `as` 运算符 (§ 7.9.10) 的当前规范中, 对于表达式 `e as T` 最后一点表明, 如果从 `e` 的编译时类型到 `T`, 不存在有效的显式引用转换, 将会出现编译时错误。对于泛型, 这条规则稍微做了修改。如果 `e` 的编译时类型或 `T` 是一个开放类型, 在这种情况下将不会出现编译时错误; 相反, 将会执行运行时检查。

20.8.7 异常语句

对于开放类型, `throw` (§ 8.9.5) 和 `try` (§ 8.10) 的通常规则是适用的。

- 只有类型参数具有 `System.Exception` 异常 (或子类具有) 作为类约束, `throw` 语句才可以用做一个表达式, 这个表达式的类型由一个类型参数给定。
- 只有类型参数 `System.Exception` (或子类具有) 作为类约束, 在 `catch` 语句中的命名的类型才可以是一个类型参数。

20.8.8 lock 语句

`lock` 语句可以用做其类型由一个类型参数给定的表达式。如果表达式的运行时类型是一个值类型, `lock` 将没有效果 (因为对于装箱值不能有任何其他的引用)。

20.8.9 using 语句

`using` 语句 (§ 8.13) 遵循通常的规则: 表达式必须被隐式地转换到 `System.IDisposable`。如果类型参数通过 `System.IDisposable` 而约束, 那么该类型的表达式可以使用 `using` 语句。

20.8.10 foreach 语句

给定如下形式的 `foreach` 语句:

```
foreach(ElementType element in collection) statement
```

如果 `collection` 表达式是一个没有实现集合模式的类型, 但它为类型 `T` 实现了构造接口 `System.Collections.Generic.IEnumerable<T>`, 那么 `foreach` 语句的扩展将是

```
IEnumerator<T>enumerator=((IEnumerable<T>)(collection)).GetEnumerator();
try
{
    while (enumerator.MoveNext()){
        ElementType element = (ElementType)enumerator.Current;
        statement;
    }
}
finally{
    enumerator.Dispose();
}
```

20.9 查找规则修订

泛型修改了用于查找和绑定名称的某些基本规则。下面几节在考虑泛型的情况下，重新叙述了所有的基本名称查找规则。

20.9.1 命名空间和类型名称

如下内容可替换 § 3.8。

在 C# 程序中有几处上下文需要指定命名空间或者类型名称。任何形式的名称都可以由一个或多个由“.”标记分隔的标识符号组成。

namespace-name: (命名空间名称:)

namespace-or-type-name (命名空间或类型名称)

type-name: (类型名称:)

namespace-or-type-name (命名空间或类型名称)

namespace-or-type-name: (命名空间或类型名称:)

identifier type-argument-list_{opt} (标识符 类型实参列表_{可选})

namespace-or-type-name . identifier type-argument-list_{opt} (命名空间或类型名称. 标识符 类型实参列表_{可选})

命名空间名称是引用命名空间的命名空间名称或类型名称。见下面所描述的决策，命名空间名称的命名空间或类型名称必须引用一个命名空间，否则出现编译时错误。在一个命名空间名称中不能有类型实参（只有类型可以具有类型实参）。

类型名称是一个命名空间或类型名称（**namespace-or-type-name**），它引用一个类型。见下面所描述的决策，类型名称的命名空间或类型名称必须引用一个类型，否则出现编译时错误。

命名空间或类型名称的意义按如下内容确定。

- 如果命名空间或类型名称是 **I** 或 **I<A₁,...,A_N>** 的形式，其中 **I** 是一个单一标识符，并且 **<A₁,...,A_N>** 是一个可选类型实参列表。
 - ◆ 如果命名空间或类型名称出现在泛型方法声明之内，如果这个声明包括一个由 **I** 给定名称的类型参数，并且该类型参数没有指定类型实参列表，那么命名空间或类型名称引用这个类型参数。
 - ◆ 如果命名空间或类型名称出现在类型声明之内，那么对于类型 **T** 的每个实例（§ 20.1.2），以那个类型声明的实例类型开始，并继续采用每个封闭类或结构类型声明的实例类型（如果有的话）。
 - 如果 **T** 的声明包括由 **I** 指定名称的类型参数，并且没有指定类型实参列表，那么命名空间或类型名称引用这个类型参数。
 - 否则，如果 **I** 是 **T** 中可访问成员的名称，并且如果那个成员是一个带有匹配类型参数数量的类型，那么命名空间或类型名称引用类型 **T.I** 或者类型 **T.I<A₁,...,A_N>**。注意当确定命名空间或类型名称的意义时，非类型

(nontype) 成员 (常数、字段、方法、属性、索引器、运算符、实例构造函数、析构函数和静态构造函数) 和带有不同数量的类型参数的成员将会被忽略。

- ◆ 否则, 对于每个命名空间 N (以命名空间或类型名称出现的命名空间开始, 并继续使用每个封闭命名空间 (如果有的话), 然后以全局命名空间结束), 下面的步骤将会被计算, 直到一个实体被定位。
 - 如果 I 是在 N 中的命名空间中的名称, 并且没有指定类型实参列表, 那么命名空间或类型名称引用该命名空间。
 - 如果 I 是在 N 中带有匹配类型参数数量的可访问类型的名称, 那么命名空间或类型名称引用给定类型实参构造的类型。
 - 如果命名空间或类型名称出现的位置, 由 N 的命名空间声明所封闭, 则:
 - 如果命名空间声明包含一个与 I 给定的名称相关联的 `using` 别名指示符, 而 I 带有导入命名空间或类型, 并且没有指定类型实参列表, 那么这个命名空间或类型名称引用该命名空间或者类型。
 - 如果由命名空间声明的 `using` 命名空间指示符导入的命名空间, 恰好包含带有 I 给定名称的, 匹配类型参数数量的类型, 那么命名空间或类型名称引用由给定类型实参构造的这个类型。
 - 如果由命名空间声明的 `using` 命名空间指示符导入的命名空间包含多个带有 I 给定名称的, 匹配类型参数数量的类型, 那么命名空间或者类型名称是模糊的, 并且将导致错误。
- ◆ 否则, 命名空间或类型名称是未定义的, 并且出现编译时错误。
- 否则, 命名空间或者类型名称是 $N.I$ 或者 $N.I\langle A_1, \dots, A_N \rangle$ 的形式, 这里 N 是一个命名空间或类型名称, I 是一个标识符, 并且 $\langle A_1, \dots, A_N \rangle$ 是一个可选类型实参列表。 N 被作为命名空间或类型名称而首先决策。如果 N 的决策失败, 将会导致编译时错误。否则 $N.I$ 或者 $N.I\langle A_1, \dots, A_N \rangle$ 将按如下方式决策。
 - ◆ 如果 N 引用一个命名空间, 并且如果 I 是内嵌在 N 中的命名空间名称, 并且没有指定类型实参列表, 那么命名空间或类型名称引用该内嵌命名空间。
 - ◆ 如果 N 引用一个命名空间, 并且 I 是在带有匹配类型参数数量的 N 中的可访问类型的名称, 那么命名空间或类型名称引用由给定类型实参构造的那个类型。
 - ◆ 如果 N 引用类或结构类型, 并且 I 是内嵌在带有与类型参数匹配的 N 中的可访问类型的名称, 那么命名空间或者类型名称引用由给定实参构造的那个类型。
 - ◆ 否则, $N.I$ 是一个无效的命名空间名称, 并且会出现编译时错误。

20.9.2 成员查找

下面的内容可替换 § 7.3。

成员查找是一种根据在上下文中的意义来确定类型的处理过程。在一个表达式中, 成

员查找可以作为计算一个简单名称或成员访问 (§ 20.9.4) 的部分而出现。

在类型 *T* 中的名称 *N* 的成员查找按如下规则确定。

- 首先一组名为 *N* 的可访问成员被确定。
 - 如果 *T* 是一个类型参数, 那么这个集合就是名为 *N* 的可访问成员的集合的联合, 在这些类型中每一个, 都被指定作为 *T* 的类约束或接口约束, 连同在 *object* 中的名为 *N* 的可访问成员的集合。
 - 否则, 这个集合由 *T* 中的名为 *N* 的所有可访问成员组成, 包括继承成员和在 *object* 中的名为 *N* 的可访问成员。如果 *T* 是一个构造类型, 则成员的集合通过替换 § 20.5.4 中描述的类型实参而得到。包括 *override* 修饰符的成员将从集合中排除。
- 接着, 通过其他成员而被隐藏的成员将从这个集合中删除。对于在集合中的每个成员 *S.M*, 其中 *S* 是 *M* 在其中被声明的类型, 则下面的规则可适用。
 - 如果 *M* 是一个常数、字段、属性、事件或枚举成员, 那么在 *S* 的所有基类中声明的成员都将从这个集合删除。
 - 如果 *M* 是一个类型声明, 那么在 *S* 的基类型中的所有非类型声明都从集合被删除, 并且, 作为 *S* 在基类型中声明的 *M* 的所有带有相同数量类型参数的类型声明, 都将从该集合中删除。
 - 如果 *M* 是一个方法, 那么在 *S* 的基类中声明的所有非方法成员, 都将从这个集合删除, 并且, 作为 *S* 在基类型中声明的 *M* 的带有相同签名的所有方法都将从这个集合中删除。
- 接着, 通过类成员隐藏的接口成员将从该集合中删除。只有当 *T* 是一个类型参数, 并且 *T* 有类约束和多个接口约束时这一步才有效。对于在集合中的每个成员 *S.M*, 其中 *S* 是 *M* 在其中声明的类型, 如果 *S* 是一个除 *object* 外的类声明, 则下面的规则适用。
 - 如果 *M* 是一个常数、字段、属性、事件、枚举成员或类型声明, 那么在接口声明中声明的所有成员都将从这个集合中删除。
 - 如果 *M* 是一个方法, 那么在接口类型中声明的所有非方法成员都将从这个集合中删除, 并且, 与在接口中声明的 *M* 一样, 带有相同签名的所有方法都将从这个集合中删除。
- 最后, 在删除隐藏成员之后, 查找的结果将被确定。
 - 如果由单一成员组成的集合, 不是类型、方法, 那么这个成员就是查找的结果。
 - 如果集合只包含方法, 那么这组方法就是查找的结果。
 - 如果集合只包含类型声明, 那么这组类型声明在成员查找的结果中。
 - 否则, 查找是不明确的, 将会出现编译时错误。

对于类型中而不是类型参数和接口中的成员查找来说, 在接口中的成员查找是严格的单继承 (在继承链中的每个接口恰好有零个或一个直接基接口), 查找规则的影响只是派生成员隐藏带有相同名称和签名的基类成员。这种单继承查找是很明确的。成员查找中可能引起的模糊性来自于 § 13.2.5 中描述的多重继承接口。

20.9.3 简单名称

如下内容可替换 § 7.5.2。

简单名称包含一个标识符，标识符后可跟随可选的类型参数列表。

simple-name: (简单名称:)

identifier type-argument-list_{opt} (标识符 类型实参列表_{可选})

对于 I 或 I<A₁,...,A_N>形式的简单名称 (这里 I 是一个标识符, <A₁,...,A_N>是一个可选类型实参列表) 可以被按如下方式计算和分类。

- 如果简单名称出现在块内，并且如果块的局部变量声明空间包含由 I 给定名称的局部变量或参数，那么，这个简单名称引用该局部变量和参数，并且将其作为一个变量而进行分类。如果指定了类型实参列表，将会出现编译时错误。
- 如果简单名称出现在泛型方法声明体之内，并且如果该声明包含由 I 给定名称的类型参数，那么简单名称引用那个类型参数，如果指定了类型实参列表，将会出现编译时错误。
- 否则，对于以直接封闭类、结构或枚举声明的实例类型开始的类型 T 的每个实例，继续采用每个封闭外部类或结构声明的实例类型 (如果有的话)。
 - ◆ 如果 T 的声明包括由 I 给定名称的类型参数，那么，简单名称引用该类型参数。如果指定了类型实参列表，将会出现编译时错误。
 - ◆ 否则，如果在 T 中 I 的成员查找产生一个匹配，则：
 - 如果 T 是直接封闭类或结构类型的实例类型，并且查找标识一个或多个方法，结果将是一个带有与 this 表达式关联的方法组。如果指定了类型实参列表，它将用在泛型方法调用中 (§ 20.6.3)。
 - 如果 T 是直接封闭类或结构类型的实例类型，如果查找标识一个实例成员，并且引用出现在一个实例构造函数、实例方法或一个实例访问器的块之内，其结果将和 this.I 形式的成员访问相似。如果指定了类型实参，将出现编译时错误。
 - 否则，结果与 T.I 或 T.I<A₁,...,A_N>形式的成员访问相似。在这种情况下，引用实例成员的简单名称将导致编译时错误。
- 否则，对于每个命名空间 N (以命名空间或类型名字出现的命名空间开始继续采用每个封闭命名空间 (如果有的话)，并以全局命名空间结束)，下面的步骤将被计算，直到一个实体被定位。
 - ◆ 如果 I 是在 N 中的一个命名空间的名称，并且没有指定类型实参列表，那么简单名称将引用该命名空间。
 - ◆ 否则，如果 I 是 N 中可访问类型的名称而 N 带有数量匹配的类型参数，那么简单类型引用由给定类型实参构造的那个类型。
 - 如果命名空间声明包含一个关联由 I 给定名称的 using 别名指示符，这里 I 是一个导入命名空间或类型，并且没有指定类型实参列表，那么简单名称引用该命名空间或类型。

- 否则，如果由命名空间声明的 `using` 命名空间指示符导入的命名空间，恰好包含一个由 `I` 给定名称的，匹配类型参数数量的类型，那么简单名称引用由给定类型实参构造的类型。
- 否则，如果由命名空间声明的 `using` 命名空间指示符导入的命名空间，包含多个由 `I` 给定名称，匹配类型参数数量的类型，那么简单名称是不明确的，将导致编译时错误。
- 否则，由简单名称给定的名称是未定义的，将导致编译时错误。

20.9.4 成员访问

如下内容可替代 § 7.5.4。

成员访问由基本表达式或预定义类型，紧跟一个“.”标记，再紧跟一个标识符，然后是可选的类型实参列表而组成。

`member-access`: (成员访问:)

`primary-expression . identifier type-argument-listopt` (基本表达式. 标识符 类型实参列表_{可选})

`predefined-type . identifier type-argument-listopt` (预定义类型. 标识符 类型实参列表_{可选}) 预定义类型: 下列之一

`bool byte char decimal double float int long`
`object sbyte short string uint ulong ushort`

对于 `E.I` 或 `E.I<A1,...,AN>` 形式的成员访问，这里 `E` 是一个基本表达式或预定义类型，`I` 是一个标识符，并且 `<A1,...,AN>` 是一个可选类型实参列表，将按如下方式被计算和分类。

- 如果 `E` 是一个命名空间，`I` 是 `E` 中嵌套命名空间的名称，并且没有指定类型实参列表，那么结果就是这个命名空间。
- 如果 `E` 是一个命名空间，`I` 是在 `E` 中可访问类型的名称，`E` 带有数量匹配的类型参数，那么结果就是由给定类型实参构造的类型。
- 如果 `E` 是一个预定义类型或作为一个类型而被分类的基本表达式，`E` 不是类型参数，并且在 `E` 中 `I` 的成员查找产生一个匹配，那么 `E.I` 被计算，并按如下方式分类。
 - ◆ 如果 `I` 标识一个或多个类型声明，那么使用与在类型实参列表中提供的数量相同的（可能是零）类型参数来确定该类型声明，结果就是由给定类型实参而构造的类型。如果类型声明不匹配类型参数的数量，将出现编译时错误。
 - ◆ 如果 `I` 标识一个或多个方法，那么结果是没有关联的实例表达式的方法组。如果指定了类型实参列表，它将在泛型方法调用中被使用 (§ 20.6.3)。
 - ◆ 如果 `I` 标识一个静态属性、静态字段、静态事件、常数或一个枚举成员，并且如果指定了类型实参列表，将出现编译时错误。
 - ◆ 如果 `I` 标识一个静态属性，那么结果是带有无关联的实例表达式的属性访问。
 - ◆ 如果 `I` 标识一个静态字段
 - 如果字段是只读的，并且引用发生在字段被声明的类或结构的静态构造函数

- 之外，那么结果是一个值，也就是在 E 中静态字段 I 的值。
 - 否则，结果是一个变量，也就是在 E 中的静态字段 I。
- ◆ 如果 I 标识一个静态事件
 - 如果引用发生在事件被声明的类或者结构中，并且事件被声明时没有使用事件访问器声明 (event-accessor-declaration) (§ 10.7)，那么 E.I 就好像 I 是一个静态字段一样被处理。
 - 否则，结果是一个无关联的实例表达式的事件访问。
- ◆ 如果 I 标识一个常数，那么结果是值，也就是常数的值。
- ◆ 如果 I 标识一个枚举成员，那么结果是一个值，也就是枚举成员的值。
- ◆ 否则，E.I 是一个无效成员引用，并将导致编译时错误。
- 如果 E 是一个属性访问、索引器访问、变量或值，其类型是 T，并且在 T 中 I 的成员查找 (§ 7.3) 产生一个匹配，那么 E.I 按如下方式被计算和分类。
 - ◆ 首先，如果 E 是一个属性或索引器访问，那么属性或索引器访问的值将被获取 (§ 7.1)，并且 E 被重分类为值。
 - ◆ 如果 I 标识一个或多个方法，那么结果是一个带有关联的 E 的实例表达式的方法组。如果指定了类型实参列表，它将在泛型方法调用中被使用 (§ 20.6.3)。
 - ◆ 如果 I 标识一个实例属性、实例字段或实例事件，并且如果指定了类型实参列表，将产生编译时错误。
 - ◆ 如果 I 标识一个实例属性，那么结果是一个带有关联的 E 的实例表达式。
 - ◆ 如果 T 是一个类类型并且 I 标识一个类类型的实例字段，那么：
 - 如果 E 的值是 null，那么将抛出 System.NullReferenceException 异常。
 - 如果该字段是只读的，并且引用出现在字段声明的类的实例构造函数之外，那么结果是值，也就是由 E 引用的对象中 I 的值。
 - 否则，结果是变量，也就是由 E 引用的对象中字段 I。
 - ◆ 如果 T 是一个结构类型，并且 I 标识该结构类型的实例字段，那么：
 - 如果 E 是一个值，或者如果字段是只读的，并且引用出现在字段声明的结构的实例构造函数之外，那么结果是一个值，也就是由 E 给定的结构实例中字段 I 的值。
 - 否则，结果是一个变量，也就是由 E 给定结构实例中的字段 I。
 - ◆ 如果 I 标识一个实例事件
 - 如果引用出现在该事件被声明的类或结构之内，并且事件被声明时没有使用事件访问器声明，那么 E.I 就好像 I 是一个实例字段一样被处理。
 - 否则，结果是一个带有关联的 E 的实例表达式。
- 否则，E.I 是一个无效成员引用，将导致编译时错误。

20.9.5 方法调用

如下内容可替换 § 7.5.5.1 中描述方法调用的编译时处理部分。

对于 M(A)形式的方法调用的编译时处理 (其中 M 是一个方法组，可能包含一个类型

实参列表, A 是可选实参列表) 由如下步骤组成。

- 方法调用的候选集合被构造。对于每个与方法组 M 关联的方法 F:
 - ◆ 如果 F 是非泛型的, 那么当如下条件成立时 F 是候选项。
 - M 没有类型实参列表, 并且
 - 对于 A, F 是适用的 (§ 7.4.2.1)。
 - ◆ 如果 F 是泛型, 并且 M 没有类型实参列表, 当如下成立时, F 是候选项
 - 类型推断 (§ 20.6.4) 成功, 对于调用推断出类型实参的列表。
 - 一旦推断的类型实参替换了对应方法类型参数, F 的参数列表对于 A 是适用的。
 - 在替换类型实参后, F 的参数列表, 与适用的可能以其扩展形式 (§ 7.4.2.1) 在相同的类型中作为 F 而声明的非泛型方法是不同的。
 - ◆ 如果 F 是泛型, 并且 M 包括一个类型实参列表, 当如下条件成立时, F 是候选项。
 - F 和在类型实参列表中提供的一样, 具有相同数量的方法类型参数, 并且
 - 一旦, 类型实参替换为对应的方法类型参数, F 的参数列表对于 A 是可适用的 (§ 7.4.2.1)。
- 候选方法的集合被缩减到只包含从深度派生类型而来的方法: 对于在集合中的每个 C.F 方法, C 是 F 在其中声明的类型, 在 C 的基类型中声明的所有方法都被从集合中删除。
- 如果候选方法的结果集合是空的, 那么没有可适用的方法存在, 并且会出现编译时错误。如果候选方法并不是在同一类型中声明的, 方法调用将是不明确的, 并且会出现编译时错误 (后一种情况, 只可能出现在对于一个在具有多重直接基接口的接口中的方法的调用, 参见 § 13.2.5 中的描述)。
- 候选方法集合的最佳方法是使用重载决策规则 (§ 7.4.2) 标识。如果不能标识一个单一的最佳方法, 则该方法调用是不明确的, 并产生编译时错误。当执行重载决策时, 泛型方法的参数在将对应的方法类型参数替换为类型实参 (提供的或推断的) 之后将被考虑。
- 被选择的最佳方法的最后验证被执行。
 - ◆ 方法在方法组的上下文中是有效的: 如果方法是一个静态方法, 方法组必须通过类型源自于简单名称或成员访问。如果该最佳方法是一个实例方法, 则方法组必须通过一个变量或值或者基类访问源自于简单名称或成员访问。如果这些需求都不满足, 那么将会出现编译时错误。
 - ◆ 如果该最佳方法是一个泛型方法, 类型实参 (提供的或推断的) 将被针对声明在泛型方法之上的约束做出检查。如果任何类型实参不满足对应类型参数的约束, 将产生一个编译时错误。

一旦方法根据前面的步骤被选择和验证, 实际的运行时调用将根据 § 7.4.3 中的函数成员调用规则而被处理。

20.9.6 委托创建表达式

如下内容可替换 § 7.5.10.3 中委托创建表达式的编译时处理部分。

对于 `new D(E)` 形式的委托创建表达式的编译时处理（其中 `D` 是一个委托类型，`E` 是一个表达式）由如下步骤组成。

- 如果 `E` 是一个方法组：
 - ◆ 将会选择一个对应于 `E(A)` 形式的方法调用，所做修正如下。
 - `D` 的参数类型和修饰符（`ref` 或 `out`）用做实参类型和实参列表 `A` 的修饰符。
 - 在适用的测试和类型推断中，转换不予考虑。在隐式转换满足的实例中，类型要求是同一的。
 - 重载决策步骤不会执行。相反，候选的集合必须恰好包含一个与 `D` 兼容的方法（接着使用类型实参替换类型参数），并且这个方法将变成新创建委托所引用的方法。如果没有匹配的方法存在，或有多个匹配的方法存在，将发生编译时错误。
 - ◆ 如果被选择的方法是一个实例方法，与 `E` 关联的实例表达式将确定委托的目标对象。
 - ◆ 结果是一个 `D` 类型的值，也就是引用所选择的方法和目标对象的新创建委托。
- 否则，`E` 是一个委托类型的值。
 - ◆ `D` 和 `E` 必须兼容；否则将出现编译时错误。
 - ◆ 结果是 `D` 类型的值，也就是像 `E` 一样引用相同的调用列表的新创建的委托。
- 否则，委托创建表达式是无效的，并且将出现编译时错误。

20.10 右移语法改变

泛型使用“<”和“>”字符分隔类型参数和类型实参（与 C++ 的模板语法相似）。构造类型有时可嵌套，如 `List<Nullable<int>>`，但使用这种构件有一些微妙的语法问题：词法将组合这个构件的最后两个标记“>>”（右移运算符），而不是产生句法需要的两个“>”标记。尽管一个可能的解决方案是在两个“>>”中放入空格，但还是容易引起混淆，并没有增加程序的简洁性。

为了让这些中性的构件维持简单的词法，将“>>”和“>>=”标记从词法中删除了，代之的是 `right-shift` 和 `right-shift-assignment` 产生式。

运算符或标点：以下之一

```
{ } [ ] ( ) . , : ;
+ - * / % & | ^ ! ~
= < > ? ++ -- && || == ->
!= <= >= += -= *= /= %= &= |=
^= << <<=
```

`right-shift`: (右移:)

> >

right-shift-assignment: (右移赋值)

> >=

与句法中的其他产生式不同，在右移和右移赋值产生式的标记之间不允许任何种类的字符存在（即使是空格也不允许）。

下面的产生式被使用右移或右移赋值进行了修改。

shift-expression: (移位表达式:)

additive-expression (附加表达式)

shift-expression << additive-expression (移位表达式 << 附加表达式)

shift-expression right-shift additive-expression (移位表达式 right-shift 附加表达式)

assignment-operator: (赋值运算符:)

=

+=

-=

*=

/=

%=

&=

|=

^=

<<=

right-shift-assignment

overloadable-binary-operator: (可重载二元运算符:)

+

-

*

/

%

&

|

^

<<

right-shift

==

!=

>

<

>=

<=

第 21 章 匿名方法

21.1 匿名方法表达式

匿名方法表达式（`anonymous-method-expression`）定义了匿名方法（`anonymous method`），并得到引用该方法的一个具体值。

`primary-no-array-creation-expression`:（基本非数组创建表达式:）

...

`anonymous-method-expression`（匿名方法表达式）

`anonymous-method-expression`:

`delegate` `anonymous-method-signature`_{opt} `block`（匿名方法表达式:

`delegate` 匿名方法签名 _{可选} 块）

`anonymous-method-signature`:

（ `anonymous-method-parameter-list`_{opt} ）（匿名方法签名: 匿名方法参数列表 _{可选}）

`anonymous-method-parameter-list`:

`anonymous-method-parameter`

`anonymous-method-parameter-list` , `anonymous-method-parameter`（匿名方法参数列表: 匿名方法参数 匿名方法参数列表）

`anonymous-method-parameter`:

`parameter-modifier`_{opt} `type` `identifier`（匿名方法参数: 参数修饰符 _{可选} 类型 标识符）

匿名方法表达式是具有特定转换规则（§21.3）的值。这个值没有类型，但它可以被隐式转换到与之兼容的委托类型。

匿名方法表达式为参数、局部变量和常数定义了一个新的声明空间，并且为标签（§3.3）定义了一个新的声明空间。

21.2 匿名方法签名

可选的**匿名方法签名**（`anonymous-method-signature`）为该匿名方法定义了正式参数的名字和类型。匿名方法的参数作用域为匿名方法的块（`block`）。如果一个匿名方法的参数的名称，与作用域包含这个匿名方法表达式的局部变量、局部常量或参数的名称匹配，那么将产生一个编译时错误。

如果一个匿名方法表达式具有匿名方法签名，那么与之兼容的委托类型集合将被限制为那些具有相同顺序 (§21.3) 相同参数类型和修饰符的委托类型集合。如果匿名方法表达式不具有匿名方法签名，那么与之兼容的委托类型将被限制为那些没有 out 参数的委托类型集合。

请注意，匿名方法签名不能包含特性或者参数数组。不过，匿名方法签名可以和其参数列表包含参数数组的委托类型兼容。

21.3 匿名方法转换

匿名方法表达式是一个无类型的值。匿名方法表达式可以用于委托创建表达式中。匿名方法表达式的所有其他合法的使用取决于在此定义的隐式转换。

匿名方法表达式与任何与之兼容的委托类型之间都存在隐式转换。如果 D 是一个委托类型，而 A 是一个匿名方法表达式，那么如果下面的条件满足的话，D 就与 A 兼容。

- 首先，D 的参数类型与 A 兼容：
 - 如果 A 不包含匿名方法签名，那么 D 可以有零个或多个任意类型的参数，前提是 D 的参数没有任何 out 参数修饰符。
 - 如果 A 具有匿名方法签名，那么 D 必须具有相同数量的参数，A 的每个参数与 D 的对应参数必须具有相同的类型，并且在 A 上的每个参数的 ref 或 out 修饰符的存在与否，都必须与 D 的对应参数相匹配。D 的最后一个参数是否是参数数组和 D 与 A 的兼容性无关。
- 其次，D 的返回类型必须与 A 兼容，对于这些规则，不考虑 A 包含任何其他匿名方法块的情况。
 - 如果 D 使用 void 声明返回类型，那么包含在 A 中的任何返回语句都不应该指定表达式。
 - 如果 D 使用类型 R 声明返回类型，那么包含在 A 中的任何返回语句的都必须指定一个可以隐式转换 (§6.1) 到 R 的表达式。并且，A 的块的结束点必须是不可达的。

除了从匿名方法到与之兼容的委托类型的隐式转换之外，匿名方法不存在任何其他转换，即便是对于 object 类型也是如此。

下面的例子说明了这些规则：

```
delegate void D(int x);
D d1 = delegate { }; // Ok
D d2 = delegate() { }; // 错误，签名不匹配
D d3 = delegate(long x) { }; // 错误，签名不匹配
D d4 = delegate(int x) { }; // Ok
D d5 = delegate(int x) { return; }; // Ok
D d6 = delegate(int x) { return x; }; // 错误，返回类型不匹配
delegate void E(out int x);
E e1 = delegate { }; // 错误 e 具有输出参数
E e2 = delegate(out int x) { x = 1; }; // Ok
E e3 = delegate(ref int x) { x = 1; }; // 错误，签名不匹配
delegate int P(params int[] a);
```

```

P p1 = delegate { }; // 错误, 块的结束点可达
P p2 = delegate { return; }; // 错误, 返回类型不匹配
P p3 = delegate { return 1; }; // Ok
P p4 = delegate { return "Hello"; }; // 错误, 返回类型不匹配
P p5 = delegate(int[] a) { // Ok
    return a[0];
};
P p6 = delegate(params int[] a) { // 错误, 具有 params 修饰符
    return a[0];
};
P p7 = delegate(int[] a) { // 错误, 返回类型不匹配
    if (a.Length > 0) return a[0];
    return "Hello";
};
delegate object Q(params int[] a);
Q q1 = delegate(int[] a) { // Ok
    if (a.Length > 0) return a[0];
    return "Hello";
};

```

委托创建表达式[delegate-creation-expression] (§7.5.10.3) 可用做将匿名方法转换到一个委托类型的替代语法。如果用做委托创建表达式的实参的表达式是一个匿名方法表达式, 那么匿名方法将使用上面定义的隐式转换规则转换到给定的委托类型。例如, 如果 D 是一个委托类型, 那么表达式

```
new D(delegate { Console.WriteLine("hello"); })
```

等价于

```
(D) delegate { Console.WriteLine("hello"); }
```

21.4 匿名方法块

匿名方法表达式的块遵循下列规则:

- 如果匿名方法包含签名, 那么在签名中指定的参数在块内是有效的。如果匿名方法不具有签名, 它可以被转换为具有参数的委托类型 (§21.3), 但参数在块内不可访问。
- 除了在最接近的封闭匿名方法签名中指定的 ref 和 out 参数 (如果有的话) 以外, 对于块来说, 访问 ref 或者 out 参数将导致编译时错误。
- 当 this 的类型是一个结构类型时, 对于块来说, 访问 this 将导致编译时错误。无论该访问是显式的 (像 this.x) 还是隐式的 (像对于在结构实例的成员中的 x), 情况都是如此。该规则只是禁止此类访问方式, 并不影响在结构中成员查找的结果。
- 块可以访问匿名方法的外部变量 (§21.5)。当匿名方法表达式被求值 (§21.6) 的时候, 对于外部变量的访问, 将会引用活动的 (active) 变量的实例。
- 对于块来说, 包含一个其目标在块之外, 或一个内嵌的匿名方法的块之内的 goto 语句、break 语句或 continue 语句, 将导致编译时错误。
- 在块内的 return 语句, 将从最接近的封闭匿名方法调用中返回控制权, 而不是从封闭函数成员中返回。在 return 语句中指定的表达式必须与某个委托类型兼容, 而最接近的匿名方法表达式将被转换到该委托类型 (§21.3)。

除了通过匿名方法表达式的求值和调用之外，执行一个匿名方法的程序块，并没有明确地限制。具体而言，编译器可以通过合成一个或多个命名方法或类型来实现匿名方法。任何此类合成的元素的名字，必须保留在编译器的使用空间中：名字必须保留两个连续下划字符。

21.5 外部变量

作用域包含匿名方法表达式的任何局部变量、值参数和参数数组，都被称为匿名方法表达式的外部变量（outer variable）。在类的实例函数成员中，`this` 值被认为是一个值参数，它也是包含在函数成员内的任何匿名方法表达式的外部变量。

21.5.1 捕获外部变量

当外部变量被匿名方法引用时，就可以说这个外部变量被匿名方法所**捕获**（captured）了。通常，局部变量的生存期被限制为它所关联的程序块或语句的执行区（§5.1.7）。但被捕获的外部变量的生存期将至少被延长，直到引用匿名方法的委托可以被垃圾回收时为止。

示例：

```
using System;
delegate int D();
class Test
{
    static D F() {
        int x = 0;
        D result = delegate { return ++x; }; //❶
        return result;
    }

    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}
```

局部变量 `x` 被匿名方法所捕获，并且 `x` 的生存期至少被延长，直到从 `F` 中返回的委托可以被垃圾回收为止（在这里，这一点直到程序结束才满足）。既然匿名方法的每次调用都在 `x` 的相同实例上进行操作，该示例输出的结果为：

```
1
2
3
```

当局部变量或值参数被匿名方法所捕获时，该局部变量和值参数将不再被认为是固定（fixed）变量（§18.3），相反它成了可移动（moveable）变量。因此，任何想取得被捕获

^❶ 这里原文漏掉了分号，在 .NET Framework version 1.2.30703 环境下编译时无法通过。

的外部变量地址的不安全代码都必须首先使用 `fixed` 语句固定该变量。

21.5.2 局部变量实例化

当程序执行到变量的作用域时，就认为局部变量被**实例化**（**instantiated**）了。例如，当下面的方法被调用时，局部变量将被三次实例化和初始化——对于循环中的每次迭代都有一次。

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

但是，如果将 `x` 的声明移出循环之外，则对于 `x` 只会产生一次实例化。

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

通常，我们无法确切地看到一个局部变量多久被实例化一次——因为实例化的生命期被拆散（**disjoint**）了，可能的情况是，每次实例化都只是使用相同的存储位置。然而当一个匿名方法捕获一个局部变量时，实例化的影响将变得很明显。例如，示例：

```
using System;
delegate void D();
class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = delegate { Console.WriteLine(x); };
        }
        return result;
    }
    static void Main() {
        foreach (D d in F()) d();
    }
}
```

产生如下输出：

```
1
3
5
```

但如果将 `x` 的声明移到循环之外

```
static D[] F() {
```

```

        D[] result = new D[3];
        int x;
        for (int i = 0; i < 3; i++) {
            x = i * 2 + 1;
            result[i] = delegate { Console.WriteLine(x); };
        }
        return result;
    }

```

其输出如下：

```

5
5
5

```

请注意依据相等运算 (§21.7)，在 F 的新版本中创建的三个委托是等价的。并且，允许编译器（但不必须）将三次实例化优化为一个单一的委托实例 (§21.6)。

你可以让匿名方法委托共享某些具有其他单独实例的被捕获变量。例如，如果 F 被改变：

```

static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = delegate { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}

```

这三个委托捕获了 x 的同一实例，但捕获了 y 的多个单独实例，所以输出如下：

```

1 1
2 1
3 1

```

单独的匿名方法可以捕获外部变量的相同实例。例如：

```

using System;
delegate void Setter(int value);
delegate int Getter();
class Test
{
    static void Main() {
        int x = 0;
        Setter s = delegate(int value) { x = value; };
        Getter g = delegate { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}

```

两个匿名方法捕获了局部变量 x 的同一实例，并且它们可以通过该变量“通信”。该示例输出如下：

```

5
10

```

21.6 匿名方法求值

匿名方法表达式的运行时求值产生一个引用匿名方法的委托实例，并且被捕获的外部变量的集合（可能为空）在求值时是活动的。当由匿名方法表达式所产生的委托被调用时，匿名方法体就会执行。方法体内的代码将使用由该委托引用而被捕获的外部变量执行。

由匿名方法表达式产生的委托调用列表包含一个单一入口。该委托的确切目标对象和目标方法都是未指定的。需要特别注意的是，委托的目标对象是否为 `null`，以及封闭函数成员的 `this` 值或其他对象，都是未指定的。

对于语义上相同的匿名方法表达式的求值，如果它们具有相同被捕获的外部变量集合（可能为空），可以（但不是必须）返回相同的委托实例。术语“语义上相同”用在这里，意思是说，该匿名方法的执行期在所有情况下，在给定相同实参时都产生相同的效果。这条规则允许如下的代码优化：

```
delegate double Function(double x);
class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void F(double[] a, double[] b) {
        a = Apply(a, delegate(double x) { return Math.Sin(x); });
        b = Apply(b, delegate(double y) { return Math.Sin(y); });
        ...
    }
}
```

由于两个匿名方法委托具有被捕获外部变量的相同集合（都为空），并且匿名方法在语义上是相同的，因此允许编译器产生引用同一目标方法的委托。实际上，这里允许编译器从两个匿名方法表达式返回相同的委托实例。

21.7 委托实例相等性

如下规则适用由匿名方法委托实例的相等运算符（§7.9.8）和 `object.Equals` 方法产生的结果。

- 当委托实例是由具有相同被捕获外部变量集合的语义相同的匿名方法表达式求值而产生时，它们可以（但不必须）相等。
- 当委托实例由具有语义不同的匿名方法表达式产生，或具有不同的被捕获外部变量集合时，它们绝不相等。

21.8 明确赋值

匿名方法参数的明确赋值状态与命名方法是相同的。也就是，引用参数和值参数被明确地赋初值，而输出参数不用赋初值。并且，输出参数在匿名方法正常返回之前必须被明确赋值 (§5.1.6)。

当控制转换到匿名方法表达式的程序块时，对外部变量 *v* 的明确赋值状态，与在匿名方法表达式之前的 *v* 的明确赋值状态是相同的。也就是，外部变量的明确赋值将从匿名方法表达式上下文被继承。在匿名方法程序块内，明确赋值将和在普通程序块内一样而得到演绎 (§5.3.3)。

在匿名方法表达式之后的变量 *v* 的明确赋值状态，与在匿名方法表达式之前它的明确赋值状态相同。

例如：

```
delegate bool Filter(int i);
void F() {
    int max;
    // 错误, max 没有明确赋值
    Filter f = delegate(int n) { return n < max; }
    max = 5;
    DoWork(f);
}
```

将产生编译时错误，因为 *max* 没有在匿名方法声明的地方明确赋值。示例：

```
delegate void D();
void F() {
    int n;
    D d = delegate { n = 1; };
    d();
    //错误, n 没有明确赋值
    Console.WriteLine(n);
}
```

也将产生一个编译时错误，因为匿名方法内 *n* 的赋值，对该匿名方法外部 *n* 的明确赋值状态没有影响。

21.9 方法组转换

与在§21.3 中描述的隐式匿名方法转换相似，从方法组 (§7.1) 到兼容的委托类型也存在隐式转换。

对于给定的方法组 *E* 和委托类型 *D*，如果允许 *new D(E)* 形式的委托创建表达式 (§7.5.10.3 和 §20.9.6)，那么就存在从 *E* 到 *D* 的隐式转换，并且转换的结果恰好等价于 *new D(E)*。

在以下示例中：

```
using System;
using System.Windows.Forms;
```

```

class AlertDialog
{
    Label message = new Label();
    Button okButton = new Button();
    Button cancelButton = new Button();
    public AlertDialog() {
        okButton.Click += new EventHandler(OkClick);
        cancelButton.Click += new EventHandler(CancelClick);
        ...
    }
    void OkClick(object sender, EventArgs e) {
        ...
    }
    void CancelClick(object sender, EventArgs e) {
        ...
    }
}

```

构造函数用 `new` 运算符创建了两个委托实例。隐式方法组转换允许将之简化为

```

public AlertDialog() {
    okButton.Click += OkClick;
    cancelButton.Click += CancelClick;
    ...
}

```

对于所有其他隐式和显式的转换，转换运算符可以用于显式地执行一个特定转换。为此，示例：

```
object obj = new EventHandler(myDialog.OkClick);
```

可被替代为如下的形式：

```
object obj = (EventHandler)myDialog.OkClick;
```

方法组和匿名方法表达式可以影响重载决策（overload resolution），但它们并不参与类型推断。请参见§20.6.4 获取更详细的信息。

21.10 实现示例

本节以标准 C# 的构件形式描述匿名方法的可能实现。在这里描述的实现基于 Microsoft C# 编译器所采用的相同原则，但它绝不是强制性的或惟一可能的实现。

本节的后面部分给出了几个示例代码，中间包含了具有不同特性的匿名方法。对于每个示例，我们将提供使用惟一标准 C# 构件的代码的对应转换。在这些例子中，标识符 `D` 假定表示如下委托类型。

```
public delegate void D();
```

匿名方法的最简形式就是没有捕获外部变量的那个形式。

```

class Test
{
    static void F() {
        D d = delegate { Console.WriteLine("test"); };
    }
}

```

这段代码可被转换到一个引用编译器生成的静态方法的委托实例，而匿名方法的代码将会放入到该静态方法中：

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }
    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

在下面的示例中，匿名方法引用 `this` 的实例成员：

```
class Test
{
    int x;
    void F() {
        D d = delegate { Console.WriteLine(x); };
    }
}
```

以上这段代码可以被转换到由编译器生成的包含匿名方法代码的实例方法：

```
class Test
{
    int x;
    void F() {
        D d = new D(__Method1);
    }
    void __Method1() {
        Console.WriteLine(x);
    }
}
```

在这个例子中，匿名方法捕获了一个局部变量。

```
class Test
{
    void F() {
        int y = 123;
        D d = delegate { Console.WriteLine(y); };
    }
}
```

该局部变量的生存期现在必须至少延长到匿名方法委托的生存期为止。这可以通过将局部变量“提升 (lifting)”为编译器生成的 (compiler-generated) 类的字段来完成。局部变量实例化 (§21.5.2) 对应于创建一个编译器生成的类的实例，而访问局部变量将对应于访问编译器生成的类实例的一个字段。并且，匿名方法将成为编译器生成类的实例方法：

```
class Test
{
    void F() {
        __locals1 __locals1= new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }
}
```

```

class __Locals1
{
    public int y;
    public void __Method1() {
        Console.WriteLine(y);
    }
}

```

最后，如下匿名方法将捕获 `this` 及具有不同生存期的两个局部变量：

```

class Test
{
    int x;
    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = delegate { Console.WriteLine(x + y + z); };
        }
    }
}

```

在这里，编译器将为每个语句块生成类，在这些语句块中局部变量将被捕获，而在不同块中的局部变量将有独立的生存期。

`__Locals2` 的实例，编译器为内部语句块生成的类，包含局部变量 `z` 和引用 `__Locals1` 实例的字段。`__Locals1` 的实例，编译器为外部语句块生成的类，包含局部变量 `y` 和引用封闭函数成员的 `this` 的字段。通过这些数据结构，你可以通过 `__Locals2` 的实例到达所有被捕获的局部变量，并且匿名方法的代码可以作为那个类的实例方法而实现：

```

class Test
{
    void F() {
        __locals1 __locals1= new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __locals2 __locals2= new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }
    class __Locals1
    {
        public Test __this;
        public int y;
    }
    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;
        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}

```

第 22 章 迭代器

22.1 迭代器块

迭代器块就是产生值的有序序列的语句块 (§8.2)。迭代器块通过一个或多个 `yield` 语句区别于常规语句块。

- `yield return` 语句产生迭代的下一个值。
- `yield break` 语句指明迭代完成。

迭代器块可以用做一个方法体 (`method-body`)、运算符体 (`operator-body`)、访问器体 (`accessor-body`)，前提是对应函数成员的返回类型是枚举器接口 (§22.1.1) 之一或者可枚举接口 (§22.1.2) 之一。

迭代器块在 C# 语法中不是独特的元素。它们在几个方面受到限制，并且主要作用在函数成员声明的语义上，它们在语法上只是语句块而已。

当一个函数成员使用迭代器块实现时，该函数成员的正式参数列表指定任何 `ref` 或 `out` 参数将导致编译时错误。

`return` 语句出现在迭代器块中将导致编译时错误（但 `yield return` 语句是允许的）。

在迭代器块中包含不安全上下文 (§18.1) 将导致编译时错误。即使是当迭代器声明内嵌在不安全上下文中，迭代器块也总是定义为一个安全上下文。

22.1.1 枚举器接口

枚举器接口 (`enumerator interface`) 是 `System.Collections.IEnumerator` 接口和 `System.Collections.Generic.IEnumerator<T>` 的所有实例。在本章，这些接口将相应地作为 `IEnumerator` 和 `IEnumerator<T>` 而被引用。

22.1.2 可枚举接口

可枚举接口 (`enumerable interface`) 是 `System.Collections.IEnumerable` 接口和 `System.Collections.Generic.IEnumerable<T>` 的所有实例。在本章，这些接口将相应地作为 `IEnumerable` 和 `IEnumerable<T>` 而被引用。

22.1.3 `yield` 类型

迭代器块生成具有相同类型的所有值的序列。该类型被称为迭代器块的 **`yield` 类型**

(yield type)。

- 通常用于实现返回 IEnumerator 或 IEnumerable 的函数成员的迭代器块的 yield 类型是 object。
- 通常用于实现返回 IEnumerator<T>或 IEnumerable<T>的函数成员的迭代器块的 yield 类型是 T。

22.1.4 this 访问

在类的实例成员的迭代器块内，this 表达式是值。该值的类型就是类类型，在这个类型内可以采用这种用法，这个值就是成员被调用时对对象的引用。

在结构的实例成员的迭代器块内，this 表达式被当做一个变量。该变量的类型就是结构类型，在这个结构内它可以采用这种用法。该变量表示一个成员被调用时的对应结构的一个拷贝。在结构实例成员的迭代器块内，this 变量的行为就好像是结构类型的一个值参数。

22.2 枚举对象

当返回枚举器接口类型的函数成员使用迭代器块实现时，调用函数成员并不会立即执行迭代器块中的代码。相反，将创建和返回枚举器对象 (enumerator object)。该对象封装了在迭代器块中指定的代码，当枚举器对象的 MoveNext 方法被调用时，迭代器块中的代码就会执行。枚举器对象有如下的特征：

- 它实现了 IEnumerator 和 IEnumerator<T>, T 是迭代器块的 yield 类型(产生类型)。
- 它实现了 System.IDisposable。
- 它使用实参值 (如果有的话) 的拷贝和传递给函数成员的实例值进行初始化。
- 它有 4 个潜在的状态 before, running, suspended 和 after, 并且它在 before 状态之前被初始化。

枚举器对象通常是一个编译器生成的枚举器类实例，它封装了迭代器语句块中的代码，并且实现了枚举器接口，但其他实现方法也是可以的。如果一个枚举器类是由编译器生成的，这个类将会是内嵌的，在包含函数成员类中，类将具有私有可访问性，并且该类具有一个保留为编译器所用的名称 (§2.4.2)。

枚举器对象可以实现比在此指定的更多接口。

下面几节描述了由 IEnumerable 和 IEnumerable<T>接口实现的 MoveNext, Current 和 Dispose 成员的确切行为，这两个接口由枚举器对象提供。

请注意，枚举器对象并不支持 IEnumerator.Reset 方法。调用该方法将会抛出 System.NotSupportedException 异常。

22.2.1 MoveNext 方法

枚举器对象的 `MoveNext` 方法封装了迭代器块的代码。调用 `MoveNext` 方法将执行迭代器内的代码，并将枚举对象的 `Current` 属性设置为适当的值。由 `MoveNext` 方法执行的精确动作，取决于当 `MoveNext` 方法被调用时枚举器对象的状态。

- 如果枚举器对象的状态是 `before`，调用 `MoveNext`，则：
 - 将把状态改为 `running`。
 - 将把迭代器块的参数（包括 `this`）初始化为，当枚举器对象被初始化而保存的实参值和实例值。
 - 从开始执行迭代器块直到执行被中断（如接下来所述）。
- 如果枚举器对象的状态是 `running`，调用 `MoveNext` 的结果是未指定的。
- 如果枚举器对象的状态是 `suspended`，调用 `MoveNext`，则：
 - 将把状态改为 `running`。
- 恢复所有局部变量和参数（包括 `this`）的值为迭代器最后一次挂起（`suspended`）时执行状态所保留的值。请注意，由这些变量所引用的任何对象的内容，都可能因为前一次对 `MoveNext` 的调用而改变。
 - 在引发执行挂起的 `yield return` 语句之后重新开始执行迭代器块，并且这个状态会继续，直到执行被中断（如接下来所述）。
- 如果枚举器对象的状态是 `after`，那么调用 `MoveNext` 将返回 `false`。

当 `MoveNext` 执行迭代器块时，有 4 种方法可以中断执行：通过一个 `yield return` 语句，通过一个 `yield break` 语句，通过到达迭代器块的结束点，以及通过一个异常被抛出，并被传播到迭代器块之外。

- 当遇到一个 `yield return` 语句（§22.4）时，将会发生如下情况：
 - 在该语句中被给定的表达式将被求值，隐式地转换到 `yield` 类型，并被赋值给枚举对象的 `Current` 属性。
 - 迭代器体的执行将被挂起。所有局部变量的值和参数（包括 `this`）被保存，该 `yield return` 语句的位置也被保存。如果 `yield return` 语句在一个或多个 `try` 块之内，与之关联的 `finally` 块在此时将不会执行。
 - 枚举器对象的状态被改为 `suspended`。
 - `MoveNext` 方法对调用方返回 `true`，表明迭代器成功前进到下一个值。
- 当遇到 `yield break` 语句（§22.4）时，将会发生如下情况：
 - 如果 `yield break` 语句在一个或多个 `try` 块之内，与之关联的 `finally` 语句将被执行。
 - 枚举器对象的状态被改为 `after`。
 - `MoveNext` 方法对调用方返回 `false`，表明迭代已经完成。
- 当遇到迭代器块的结束点时，将会发生如下情况：
 - 枚举器对象的状态被改为 `after`。
 - `MoveNext` 方法对调用方返回 `false`，表明迭代已经完成。

- 当一个异常被抛出并被传播到迭代器块之外时，将会发生如下情况：
 - 在迭代器块之内将会由于异常传播(exception propagation)而执行合适的 finally 块。
 - 枚举器对象的状态被改为 after。
 - 对于 MoveNext 方法的调用方来说，异常传播将会继续。

22.2.2 Current 属性

枚举器对象的 Current 属性受到迭代器块的 yield return 语句的影响。

当枚举器对象处于 suspended 状态时，Current 的值就是最后一次调用 MoveNext 时被设置的值。当枚举器对象处于 before, running 或 after 状态时，访问 Current 的所得结果是未指定的。

对于一个具有非 object 类型的 yield 类型迭代器块，通过枚举器对象的 IEnumerable 实现访问 Current 所得实现，对应于通过枚举器对象的 IEnumerator<T>访问 Current 所得实现，并将结果转换到 object 类型。

22.2.3 Dispose 方法

Dispose 方法通过将枚举器对象的状态置为 after，以清理迭代结果。

- 如果枚举器对象的状态是 before，调用 Dispose 将改变其状态为 after。
- 如果枚举器对象的状态是 running，调用 Dispose 的结果是为指定的。
- 如果枚举器对象的状态是 suspended，调用 Dispose 则会：
 - 改变其状态为 running。
 - 执行 finally 块，就好像最后执行的 yield return 语句是一个 yield break 语句。如果这会导致一个异常被抛出并传播到迭代器体之外，则枚举器对象的状态将被置为 after，并且该异常将被传播给 Dispose 方法的调用方。
 - 改变其状态为 after。
- 如果枚举器对象的状态为 after，则调用 Dispose 没有效果。

22.3 可枚举对象

当返回一个可枚举接口类型的函数成员使用迭代器块实现时，调用函数成员不会立即执行迭代器块代码。相反，一个**可枚举对象 (enumerable object)**将被创建并返回。可枚举对象的 GetEnumerator 方法返回一个枚举器对象，它封装了在迭代器块中指定的代码，当枚举器对象的 MoveNext 方法被调用时，将触发送代器块代码的执行。可枚举对象具有如下特征。

- 它实现了 IEnumerable 和 IEnumerable<T>接口，这里 T 是迭代器块的 yield type 类型。

- 它使用实参值的拷贝进行初始化（如果有的话），并将实例值传递给函数成员。

可枚举对象通常是一个由编译器生成的可枚举类的实例，该类封装了迭代器块的代码，并实现了可枚举接口，但其他实现方法也是可以的。如果可枚举类由编译器生成，该类将内嵌在包含函数成员的类中，并具有私有可访问性，以及一个为编译器所保留使用的名字 (§2.4.2)。

可枚举对象可以实现比在此说明的更多的接口。具体而言，可枚举对象也可以实现 `IEnumerator` 和 `IEnumerator<T>` 接口，这使得它既可以作为可枚举对象又可作为枚举器对象。在该实现类型中，对可枚举对象的 `GetEnumerator` 方法的首次调用，将返回可枚举对象自身。对于该可枚举对象的 `GetEnumerator` 方法的后续调用（如果有的话）将会返回可枚举对象的一个拷贝。因此，每次返回的枚举器将有它自己的状态，并且在一个枚举器中所做的改变不会影响另一个枚举器。

可枚举对象提供了 `IEnumerable` 和 `IEnumerable<T>` 接口的 `GetEnumerator` 方法的一个实现。这两个 `GetEnumerator` 方法共享一个得到并返回一个有效的枚举器对象的公共实现。

枚举器对象使用可枚举对象被初始化时所使用的实参值和实例值进行初始化，另一方面，枚举器对象函数将如 §22.2 中所述。

22.4 yield 语句

`yield` 语句用于迭代器块以产生一个枚举器对象的值，或表明迭代的结束。

embedded-statement: (嵌入语句)

```
...
yield-statement (yield 语句)
yield-statement: (yield 语句)
    yield return expression ;
    yield break ;
```

为了确保与现存程序的兼容性，`yield` 并不是一个保留字，并且只有在紧邻 `return` 或 `break` 关键词之前使用 `yield` 才具有特别的意义。而在其他上下文中，它可以用做标识符。

`yield` 语句所能出现的地方有几个限制，如下所述。

- (两种形式的) `yield` 语句出现在方法体、运算符体和访问器体之外时，将导致编译时错误。
- (两种形式的) `yield` 语句出现在匿名方法之内时，将导致编译时错误。
- (两种形式的) `yield` 语句出现在 `try` 语句的 `finally` 从句中时，将导致编译时错误。
- `yield return` 语句出现在包含 `catch` 子语句的任何 `try` 语句中任何位置时，将导致编译时错误。

下面示例展示了 `yield` 语句的一些有效和无效用法。

```
delegate IEnumerable<int> D();
IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;           // Ok
        yield break;             // Ok
    }
```

```

    }
    finally {
        yield return 2;           // 错误, yield 在 finally 中
        yield break;             // 错误, yield 在 finally 中
    }
    try {
        yield return 3;           // 错误, yield return 在 try...catch 中
        yield break;             // Ok
    }
    catch {
        yield return 4;           // 错误, yield return 在 try...catch 中
        yield break;             // Ok
    }
    D d = delegate {
        yield return 5;           // 错误, yield 在匿名方法中
    };
}
int MyMethod() {
    yield return 1;              // 错误, 迭代器块的错误返回类型
}

```

从 `yield return` 语句中表达式类型到迭代器的 `yield` 类型 (§22.1.3), 必须存在隐式转换 (§6.1)。

`yield return` 语句按如下方式执行。

- 在语句中给出的表达式将被求值 (evaluated), 隐式地转换到 `yield` 类型, 并被赋给枚举器对象的 `Current` 属性。
- 迭代器块的执行将被挂起。如果 `yield return` 语句在一个或多个 `try` 块中, 与之关联的 `finally` 块此时将不会执行。
- 枚举器对象的 `MoveNext` 方法对调用方返回 `true`, 表明枚举器对象成功前进到下一个项。

对枚举器对象的 `MoveNext` 方法的下一次调用, 重新从迭代器块挂起的地方开始执行。

`yield break` 语句按如下方式执行。

- 如果 `yield break` 语句被包含在一个或多个带有 `finally` 块的 `try` 块内, 初始控制权将转移到最里面的 `try` 语句的 `finally` 块。当控制到达 `finally` 块的结束点后, 控制将会转移到下一个最近的 `try` 语句的 `finally` 块。这个过程会一直重复, 直到所有内部的 `try` 语句的 `finally` 块都被执行。
- 控制返回到迭代器块的调用方。这可能是由于枚举器对象的 `MoveNext` 方法或 `Dispose` 方法。

由于 `yield break` 语句无条件地将控制转移到别处, 因此 `yield break` 语句的结束点将永远是不可达的。

对于一个 `yield return expr` 形式的 `yield return` 语句 `stmt`:

- 像 `stmt` 开始一样, 在 `expr` 的开头变量 `v` 具有明确的赋值状态。
- 如果在 `expr` 的结束点 `v` 被明确赋值, 那它在 `stmt` 的结束点也将被明确赋值; 否则, 在 `stmt` 结束点将不会被明确赋值。

22.5 实现示例

本节以标准C#构件的形式描述了迭代器的可能实现。此处描述的实现基于与Microsoft C#编译器相同的原理，但这绝不是强制的或惟一可能的实现。

下面的 `Stack<T>` 类使用迭代器实现了 `GetEnumerator` 方法。该迭代器依序枚举了堆栈中从顶到底的元素。

```
using System;
using System.Collections;
using System.Collections.Generic;
class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;
    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }
    public T Pop() {
        T result = items[--count];
        items[count] = T.default;
        return result;
    }
    public IEnumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```

`GetEnumerator` 方法可以被转换到编译器生成的枚举器类的实例，该类封装了迭代器块中的代码，如下所示：

```
class Stack<T>: IEnumerable<T>
{
    ...
    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }
    class __Enumerator1: IEnumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;
        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }
        public T Current {
            get { return __current; }
        }
    }
}
```

```

        object IEnumerator.Current {
            get { return __current; }
        }
        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
__loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
__state1:
            --i;
            goto __loop;
__state2:
            __state = 2;
            return false;
        }
        public void Dispose() {
            __state = 2;
        }
        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}

```

在上面的转换中，迭代器块之内的代码被转换成状态机，并被放置在枚举器类的 `MoveNext` 方法中。此外，局部变量 `i` 被转换成枚举器对象的一个字段，因此在 `MoveNext` 的调用过程中可以持续存在。

下面的示例打印一个简单的从整数 1 到整数 10 的乘法表。该示例中 `FromTo` 方法返回一个可枚举对象，并且使用迭代器实现。

```

using System;
using System.Collections.Generic;
class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }
    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}

```

`FromTo` 方法可被转换成编译器生成的可枚举类的实例，该类封装了迭代器块中的代码，如下所示：

```
using System;
```

```
using System.Threading;
using System.Collections;
using System.Collections.Generic;
class Test
{
    ...
    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }
    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, IEnumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;
        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }
        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }
        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }
        public int Current {
            get { return __current; }
        }
        object IEnumerator.Current {
            get { return __current; }
        }
        public bool MoveNext() {
            switch (__state) {
                case 1:
                    if (from > to) goto case 2;
                    __current = from++;
                    __state = 1;
                    return true;
                case 2:
                    __state = 2;
                    return false;
                default:
                    throw new InvalidOperationException();
            }
        }
        public void Dispose() {
            __state = 2;
        }
        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

```
    }  
}
```

这个可枚举类实现了可枚举接口和枚举器接口，这使得它成为可枚举的或枚举器。当 `GetEnumerator` 方法被首次调用时，将返回可枚举对象本身。后续可枚举对象的 `GetEnumerator` 调用（如果有的话）都返回可枚举对象的拷贝。因此，每次返回的枚举器都有其自身的状态，改变一个枚举器将不会影响另一个。`Interlocked.CompareExchange` 方法用于确保线程安全操作。

`from` 和 `to` 参数被转换为可枚举类的字段。由于 `from` 在迭代器块内被修改，所以引入另一个 `__from` 字段来保存在每个枚举中给定的 `from` 的初始值。

如果当 `__state` 是 0 时 `MoveNext` 被调用，则该方法将抛出 `InvalidOperationException` 异常。这将防止没有先调用 `GetEnumerator`，而将可枚举对象作为枚举器使用的现象发生。

第 23 章 不完整类型

23.1 不完整类型声明

新类型修饰符 `partial` 用于在多个部分中定义一个类型。为了确保和现存程序的兼容性，这个修饰符和其他修饰符（比如 `get` 和 `set`）是不同的，它不是一个关键字，并且它必须紧邻在关键字 `class`，`struct` 或者 `interface` 之前出现。

```
class-declaration (类声明)
    attributesopt    class-modifiersopt    partialopt    class    identifier
    type-parameter-listopt    class-baseopt
    type-parameter-constraints-clausesopt    class-body    ;opt

(特性可选 类修饰符可选 partial可选 class 标识符
  参数列表可选 : 基类可选
  参数约束语句可选 类体; 可选)

struct-declaration: (结构声明)
    attributesopt    struct-modifiersopt    partialopt    struct    identifier
    type-parameter-listopt    struct-interfacesopt
    type-parameter-constraints-clausesopt    struct-body    ;opt

(特性可选 结构修饰符可选 partial可选 struct 标识符
  类型参数列表可选 结构接口可选 类型参数约束语句可选 结构体; 可选)

interface-declaration: (接口声明)
    attributesopt    interface-modifiersopt    partialopt    interface
    identifier    type-parameter-listopt
    interface-baseopt    type-parameter-constraints-clausesopt
    interface-body    ;opt

(特性可选 接口修饰符可选 partial可选 interface 标识符
  类型参数列表可选
  基接口可选 类型参数约束语句可选
  接口体 ; 可选)
```

不完整类型声明的每一部分都必须包含 `partial` 修饰符，并且必须在和其他部分相同的命名空间中声明。`partial` 修饰符表明该类型声明的附加部分可以存在于其他某个地方，但这种附加部分的存在并不是必需的；在一个单一类型声明中包含 `partial` 修饰符也是有效的。不完整类型的所有部分必须放在一起编译，这样它们就可以在编译时被融合。不完整类型不容许对已经被编译的类型进行扩展。

嵌套类型 (nested type) 可以通过使用 `partial` 修饰符而在多个地方声明。通常情况下, 包含类型 (也就是包含嵌套类型的类型) 也使用 `partial` 声明, 而嵌套类型的各个部分也在包含类型的不同部分中声明。

`partial` 修饰符不能用在委托或枚举声明中。

23.1.1 特性

不完整类型的特性通过以不定的 (unspecified) 顺序组合各个部分的特性而确定。如果一个特性被放在不完整类型的多个部分, 那么这等价于在该类型上多次指定该特性。例如, 这两个部分:

```
[Attr1, Attr2("hello")]
partial class A {}
[Attr3, Attr2("goodbye")]
partial class A {}
```

等价于如下声明。

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

类型参数上的特性也以相同的风格组合。

23.1.2 修饰符

当不完整类型声明包含可访问性说明 (`public`, `protected`, `internal` 和 `private` 修饰符) 时, 它必须与其他部分的访问说明一致。如果不完整类型的各个部分都不包含访问说明, 该类型将被赋予适当的默认可访问性 (§ 3.5.1)。

如果嵌套类型的一个或多个不完整声明包含 `new` 修饰符, 并且嵌套类型隐藏了一个继承成员, 将不会有任何警告 (§ 3.7.2)。

如果类的一个或多个不完整声明包含 `abstract` 修饰符, 那么这个类就是抽象的 (§ 10.1.1.1), 反之, 就是非抽象的。

如果类的一个或多个不完整声明包含 `sealed` 修饰符, 那么这个类就是密封的 (§ 10.1.1.2), 反之, 就是非密封的。

注意, 一个类不能同时既是抽象的又是密封的 (`sealed`)。

当 `unsafe` 修饰符用于一个不完整类型声明时, 只有特定的部分被认为是不安全上下文 (§ 18.1)。

23.1.3 类型参数和约束

如果泛型类型在多个部分被声明, 那么每个部分都必须说明类型参数。每个部分都必须有相同数量的类型参数, 并且对于每个类型参数必须有相同的名字和顺序。

当不完整泛型声明包含类型参数约束（**where** 从句）时，该约束必须和其他部分包含的约束一致。具体而言，包含约束的每个部分必须具有相同集合类型参数的约束，并且对于每个类型参数，类、接口和构造函数约束的集合必须是相同的。如果不完整泛型的任何部分都没有指定约束，就认为类型参数是不带约束的。

示例：

```
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}
partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}
partial class Dictionary<K,V>
{
    ...
}
```

是正确的，因为这些包含约束的部分有效地指定了类、接口的相同集合，以及相应类型参数的相同集合的构造函数约束。

23.1.4 基类

当不完整类声明包含基类说明时，它必须与包含基类说明的所有其他部分一致。如果不完整类声明的任何部分都不包含基类声明，那么基类将是 `System.Object` (§ 10.1.2.1)。

23.1.5 基接口

在多个部分中声明的类型的基接口集合，是在各个部分中指定的基接口的联合。一个特定的基接口在每个部分中只能命名一次，但可以在多个部分中命名相同的基接口。但对于任何给出的基接口成员只能有惟一的实现。

示例：

```
partial class C: IA, IB {...}
partial class C: IC {...}
partial class C: IA, IB {...}
```

中，类 `C` 的基接口是 `IA`、`IB` 和 `IC`。

通常，在接口声明的每一部分提供接口的实现；但这不是必需的。一个部分可以为声明在另一个部分中的接口提供实现：

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
```

```

    }
    partial class X: IComparable
    {
        ...
    }

```

23.1.6 成员

声明在多个部分中的类型的成员只是在各个部分中声明的成员的联合。类型声明的所有部分的内容共享相同的声明空间 (§ 3.3)，并且每个成员的作用域 (§ 3.7) 扩展到所有部分的内容。任何成员的所有可访问域总是包含封闭类型的所有部分；在一个部分中声明的 `private` 成员可以随意地在另一个部分访问。在一个类型的多个部分中声明相同的成员将导致编译时错误，除非该成员是一个带有 `partial` 修饰符的成员。

```

partial class A
{
    int x;                // 错误，不能多次声明 x
    partial class Inner   // Ok, Inner 是不完整类型
    {
        int y;
    }
}
partial class A
{
    int x;                // 错误，不能多次声明 x
    partial class Inner   // Ok, Inner 是不完整类型
    {
        int z;
    }
}

```

尽管在一个类型中成员的次序对于 C# 代码并不是太重要，但在面对其他语言和环境时却可能是很重要的。在这样的情况下，在多个部分中声明的类型内成员次序将是未定义的。

23.2 名称绑定

虽然可扩展类型的每一部分必须在相同的命名空间声明，但这些部分也可以写在不同的命名空间中。为此，对于各个部分可以使用不同的 `using` 指令 (§ 9.3)。当在一个部分中解释简单名称 (§ 7.5.2) 时，只有包含该部分的命名空间 `using` 指令被考虑。这将导致在不同部分的相同标识符表示不同的意义。

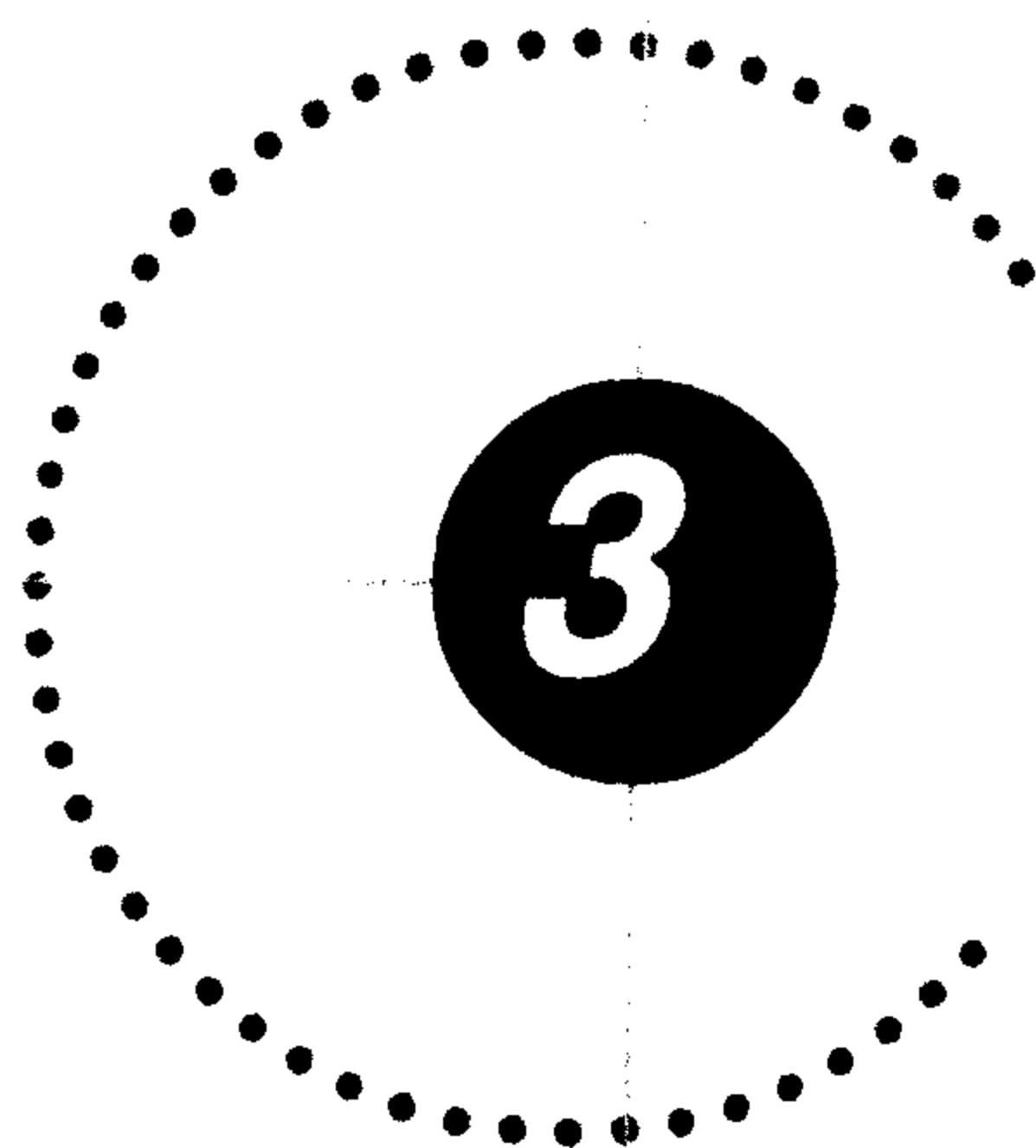
```

namespace N
{
    using List = System.Collections.ArrayList;
    partial class A
    {
        List x;            // x 具有类型 System.Collections.ArrayList
    }
}
namespace N
{

```

```
using List = Widgets.LinkedList;
partial class A
{
    List y;                // y 具有类型 Widgets.LinkedList
}
}
```

第三篇 附录



附录 A 文档注释

C#提供一种机制，使程序员可以使用含有 XML (Extensible Markup Language) 文本的特殊注释语法为他们的代码编写文档。在源代码文件中，具有某种格式的注释可用于指导某个工具根据这些注释和它们后面的源代码元素生成 XML。使用这类语法的注释称为文档注释 (documentation Comments)。这些注释后面必须紧跟用户定义类型 (如类、委托或接口) 或者成员 (如字段、事件、特性或方法)。生成 XML 的工具称为文档生成器 (documentation generator) (此生成器可以是但不一定是 C#编译器本身)。由文档生成器产生的输出称为文档文件 (documentation file)。文档文件可用做到文档查看器 (documentation viewer) 的输入；文档查看器是旨在生成类型信息及其关联文档的某种可视化显示的工具。

此规范推荐了一组在文档注释中使用的标记，但是这些标记不是必须使用的，如果需要也可以使用其他标记，只要遵循符合格式标准的 XML 规则即可。

A.1 介绍

具有特殊格式的注释可用于指导某个工具根据这些注释和它们后面的源代码元素生成 XML。这类注释是以三个斜杠 (///) 开始的单行注释，或者是以一个斜杠和两个星号 (/**) 开始的分隔注释。这些注释后面必须紧跟它们所注释的用户定义类型 (如类、委托或接口) 或者成员 (如字段、事件、特性或方法)。特性节 (§17.2) 被视为声明的一部分，因此，文档注释必须位于应用到类型或成员的特性之前。

语法：

single-line-doc-comment: (单行文档注释：)

/// input-characters_{opt} (/// 输入字符_{可选})

delimited-doc-comment: (分隔文档注释：)

/** delimited-comment-characters_{opt} */ (/** 带分隔符的注释字符_{可选} */)

在单行文档注释中，如果当前单行文档注释旁边的每个单行文档注释上的 /// 字符后跟有空白字符，则此空白字符不包括在 XML 输出中。

在分隔文档注释中，如果第二行上的第一个非空白字符是一个星号符，并且在分隔文档注释内的每行开头都重复同一种由_{可选}空白字符和星号符组成的样式，则该重复出现的样式所含的字符不包括在 XML 输出中。在此样式中，可以在星号符之前或之后包括空白字符。

示例：

```
/// <remarks>Class <c>Point</c> models a point in a two-dimensional
```

```
/// plane.</remarks>
///
public class Point
{
    /// <remarks>method <c>draw</c> renders the point.</remarks>
    void draw() {...}
}
```

文档注释内的文本必须根据 XML 规则 (<http://www.w3.org/TR/REC-xml>) 设置正确的格式。如果 XML 不符合标准格式，将生成警告，并且文档文件将包含一条注释，指出遇到错误。

尽管开发人员可自由创建他们自己的标记集，但附录 A.2 中定义了建议的标记集。某些建议的标记具有特殊含义：

- **<param>** 标记用于描述参数。如果使用这样的标记，文档生成器必须验证指定参数是否存在，以及文档注释中是否描述了所有参数。如果此验证失败，文档生成器将发出警告。
- **cref** 特性可以附加到任意标记，以提供对代码元素的引用。文档生成器必须验证此代码元素是否存在。如果验证失败，文档生成器将发出警告。查找在 **cref** 特性中描述的名称时，文档生成器必须根据源代码中出现的 **using** 语句来确定命名空间的可见性。
- **<summary>** 标记旨在标出可由文档查看器显示的有关类型或成员的额外信息。
- **<include>** 标记表示应该包含的来自外部 XML 文件的信息。

注意，文档文件并不提供有关类型和成员的完整信息（例如，它不包含任何关于类型的信息）。若要获得有关类型或成员的完整信息，必须协同使用文档文件与对实际涉及的类型或成员的反射调用。

A.2 建议的标记

文档生成器必须接受和处理任何根据 XML 规则的有效标记。下列标记提供了用户文档中常用的功能（当然，还可能有其他标记）。

表 A.1 用户文档中常用的标记及用途

标 记	章 节	用 途
<c>	A.2.1	将文本设置为类似代码的字体
<code>	A.2.2	将一行或多行源代码或程序输出设置为某种字体
<example>	A.2.3	表示所含的是示例
<exception>	A.2.4	标识出方法可能引发的异常
<include>	A.2.5	包括来自外部文件的 XML
<list>	A.2.6	创建列表或表
<para>	A.2.7	允许将结构添加到文本中
<param>	A.2.8	描述方法或构造函数的参数
<paramref>	A.2.9	确认某个单词是参数名
<permission>	A.2.10	描述成员的安全性和访问权限
<remarks>	A.2.11	描述一种类型

(续表)

标 记	章 节	用 途
<returns>	A.2.12	描述方法的返回值
<see>	A.2.13	指定链接
<seealso>	A.2.15	生成“请参见”项
<summary>	A.2.15	描述类型的成员
<value>	A.2.16	描述特性

A.2.1 <c>

此标记提供一种机制以指示用特殊字体（如用于代码块的字体）设置说明中的文本段落。对于实际代码行，请使用 <code> (§A.2.2)。

语法：

```
<c>text</c>
```

示例：

```
/// <remarks>Class <c>Point</c> models a point in a two-dimensional
/// plane.</remarks>
public class Point
{
    // ...
}
```

A.2.2 <code>

此标记用于将一行或多行源代码或程序输出设置为某种特殊字体。对于叙述中较小的代码段落，请使用 <c> (§A.2.1)。

语法：

```
<code>source code or program output</code>
```

示例：

```
/// <summary>This method changes the point's location by
/// the given x- and y-offsets.
/// <example>For example:
/// <code>
/// Point p = new Point(3,5);
/// p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

A.2.3 <example>

此标记用于在注释中插入代码示例，以说明如何使用所关联的方法或其他库成员。通常，此标记是同 <code> (§A.2.2) 标记一起使用的。

语法：

```
<example>description</example>
```

示例：

有关示例，请参见 <code> (§A.2.2)。

A.2.4 <exception>

此标记提供一种方法以说明关联的方法可能引发的异常。

语法：

```
<exception cref="member">description</exception>
```

其中：

- **member** 是成员的名称。文档生成器检查给定成员是否存在，并将 **member** 转换为文档文件中的规范元素名称。
- **description** 是对引发异常的情况的描述。

示例：

```
public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatCorruptException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}
```

A.2.5 <include>

此标记允许包含来自源代码文件外部的 XML 文档的信息。外部文件必须是符合标准格式的 XML 文档，还可以利用 XPath 表达式来指定应选择该 XML 文档中的哪些 XML 文本。然后用从外部文档中选定的 XML 来替换 <include> 标记。

语法：

```
<include file="filename" path="xpath" />
```

其中：

- filename 是外部 XML 文件的文件名。该文件名是相对于包含 include 标记的文件进行解释的（确定其完整路径名）。
- Xpath 是 XPath 表达式，用于选择外部 XML 文件中的某些 XML。

示例：

如果源代码包含了如下声明：

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

并且外部文件 docs.xml 含有以下内容：

```
<?xml version="1.0"?>
<extradoc>
  <class name="IntList">
    <summary>
      Contains a list of integers.
    </summary>
  </class>
  <class name="StringList">
    <summary>
      Contains a list of integers.
    </summary>
  </class>
</extradoc>
```

这样，输出的文档就与源代码中包含以下内容时一样：

```
/// <summary>
///   Contains a list of integers.
/// </summary>

public class IntList { ... }
```

A.2.6 <list>

此标记用于创建列表或项目表。它可以包含 <listheader> 块以定义表或定义列表的标头行（定义表时，仅需要在标头中为 term 提供一个项）。

列表中的每一项都用一个 <item> 块来描述。创建定义列表时，必须同时指定 term 和 description。但是，对于表、项目符号列表或编号列表，仅需要指定 description。

语法：

```
<list type="bullet" | "number" | "table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
  ...
  <item>
    <term>term</term>
    <description>description</description>
```

```
    </item>
  </list>
```

其中:

- **term** 是要定义的术语, 其定义位于 **description** 中。
- **description** 是项目符号列表或编号列表中的项, 或者是 **term** 的定义。

示例:

```
public class MyClass
{
    /// <remarks>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </remarks>
    public static void Main () {
        // ...
    }
}
```

A.2.7 <para>

此标记用于其他标记内, 如 **<remarks>** (§A.2.11) 或 **<returns>** (§A.2.12), 用于将结构添加到文本中。

语法:

```
<para>content</para>
```

其中, **content** 是段落文本。

示例:

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any nontrivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

A.2.8 <param>

该标记用于描述方法、构造函数或索引器的参数。

语法:

```
<param name="name">description</param>
```

其中:

- `name` 是参数名。
- `description` 是参数的描述。

示例：

```
/// <summary>This method changes the point's location to
/// the given coordinates.</summary>
/// <param name="xor"> the new x-coordinate.</param>
/// <param name="yor"> the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

A.2.9 <paramref>

该标记表示某单词是一个参数。这样，生成文档文件后经适当处理，可以用某种独特的方法来格式化该参数。

语法：

```
<paramref name="name"/>
```

其中，`name` 是参数名。

示例：

```
/// <summary>This constructor initializes the new Point to
/// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor"> the new Point's x-coordinate.</param>
/// <param name="yor"> the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

A.2.10 <permission>

该标记用于将成员的安全性和可访问性记入文档。

语法：

```
<permission cref="member">description</permission>
```

其中：

- `member` 是成员的名称。文档生成器检查给定的代码元素是否存在，并将 `member` 转换为文档文件中的规范化元素名称。
- `description` 是对成员的访问特性的说明。

示例：

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>
public static void Test(){
    // ...
}
```


}

A.2.11 <remarks>

该标记用于指定类型的概述信息（使用 <summary>, §A.2.15 描述类型的成员）。

语法：

```
<remarks>description</remarks>
```

其中，**description** 是备注文本。

示例：

```
/// <remarks>Class <c>Point</c> models a point in a  
/// two-dimensional plane.</remarks>  
public class Point  
{  
    // ...  
}
```

A.2.12 <returns>

该标记用于描述方法的返回值。

语法：

```
<returns>description</returns>
```

其中，**description** 是返回值的说明。

示例：

```
/// <summary>Report a point's location as a string.</summary>  
/// <returns>A string representing a point's location, in the form (x,y),  
/// without any leading, trailing, or embedded whitespace.</returns>  
public override string ToString() {  
    return "(" + X + ", " + Y + ")";  
}
```

A.2.13 <see>

该标记用于在文本内指定链接。使用 <seealso> (§A.2.14) 指示将在“请参见”部分中出现的文本。

语法：

```
<see cref="member"/>
```

其中，**member** 是成员的名称。文档生成器检查给定的代码元素是否存在，并将 **member** 更改为所生成的文档文件中的元素名称。

示例：

```
/// <summary>This method changes the point's location to
```

```

    /// the given coordinates.</summary>
    /// <see cref="Translate"/>
    public void Move(int xor, int yor) {
        X = xor;
        Y = yor;
    }
    /// <summary>This method changes the point's location by
    /// the given x- and y-offsets.
    /// </summary>
    /// <see cref="Move"/>
    public void Translate(int xor, int yor) {
        X += xor;
        Y += yor;
    }

```

A.2.14 <seealso>

该标记用于生成“请参见”项。使用 <see> (§A.2.13) 指示将在文本内指定链接。
语法:

```
<seealso cref="member"/>
```

其中, **member** 是成员的名称。文档生成器检查给定的代码元素是否存在, 并将 **member** 更改为所生成的文档文件中的元素名称。

示例:

```

    /// <summary>This method determines whether two Points have the same
    /// location.</summary>
    /// <seealso cref="operator==" />
    /// <seealso cref="operator!=" />
    public override bool Equals(object o) {
        //...
    }

```

A.2.15 <summary>

可以用此标记描述类型的成员。使用 <remarks> (§A.2.11) 描述类型本身。
语法:

```
<summary>description</summary>
```

其中, **description** 是关于成员的摘要描述。

示例:

```

    /// <summary>This constructor initializes the new Point to (0,0).</summary>
    public Point() : this(0,0) {
    }

```

A.2.16 <value>

可以用此标记描述要描述的特性。

语法:

```
<value> Property description </value>
```

其中, Property description 是对特性的描述。

示例:

```
// <value> Property <c>X</c> represents the point's x-coordinate.</value> public int
X
{
    get{ return x; }
    set{ x = value; }
}
```

A.3 处理文档文件

文档生成器为源代码中每个附加了“文档注释标记”的代码元素生成一个 ID 字符串。该 ID 字符串惟一地标识源元素。文档查看器利用此 ID 字符串来标识该文档所描述的对应的元数据/反射项。

文档文件不是源代码的分层表现形式;而是为每个元素生成的 ID 字符串的平面列表。

A.3.1 ID 字符串格式

文档生成器在生成 ID 字符串时遵循下列规则:

- 不在字符串中放置空白。
- 字符串的第一部分通过单个字符后跟一个冒号来标识被标识成员的种类。定义以下几种成员:

表 A .2 定义的字符及其说明

字 符	说 明
E	事件
F	字段
M	方法 (包括构造函数、析构函数和运算符)
N	命名空间
P	特性 (包括索引器)
T	类型 (如类、委托、枚举, 接口和结构)
!	错误字符串; 字符串的其他部分提供有关错误的信息。例如, 文档生成器对无法解析的链接生成错误信息。

- 字符串的第二部分是元素的完全限定名, 从命名空间的根开始。元素的名称、包含着它的类型, 以及命名空间, 都以句号分隔。如果项名本身含有句号, 则将用 # (U+0023) 字符替换。(这里假定所有元素名中都没有 “# (U+0023)” 字符。)
- 对于带有参数的方法和特性, 后面紧跟着是用括号括起来的参数列表。对于那些不带参数的方法和特性, 则省略括号。多个参数以逗号分隔。各个参数的编码与

CLI 签名相同：参数由其完全限定名表示，如下所示。例如，`int` 变成 `System.Int32`、`string` 变成 `System.String`、`object` 变成 `System.Object` 等。具有 `out` 或 `ref` 修饰符的参数在其类型名后跟有 `@` 符。对于由值传递或通过 `params` 传递的参数没有特殊表示法。数组参数表示为 [下限 : 大小 , ... , 下限 : 大小]，其中逗号数量等于秩减去一，而下限和每个维的大小（如果已知）用十进制数表示。如果未指定下限或大小，它将被省略。如果省略了某个特定维的下限及大小，则“:”也将被省略。交错数组由每个级别一个“[]”来表示。指针类型为非 `void` 的参数用类型名后面跟一个 `*` 的形式来表示。`void` 指针用类型名 `System.Void` 表示。

A.3.2 ID 字符串示例

下列各个示例分别显示一段 C# 代码，以及为每个可以含有文档注释的源元素生成的 ID 字符串：

- 类型用它们的完全限定名来表示。

```
enum Color { Red, Blue, Green }
namespace Acme
{
    interface IProcess {...}
    struct ValueType {...}
    class Widget: IProcess
    {
        public class NestedClass {...}
        public interface IMenuItem {...}
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }
}
"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
```

- 字段用它们的完全限定名来表示。

```
namespace Acme
{
    struct ValueType
    {
        private int total;
    }
    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }
    }
}
```

```
        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **ppValues;
    }
}
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- 构造函数。

```
namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}
"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- 析构函数。

```
namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}
"M:Acme.Widget.Finalize"
```

- 方法。

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }
    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }
    }
}
```

```

    public static void M0() {...}
    public void M1(char c, out float f, ref ValueType v) {...}
    public void M2(short[] x1, int[,] x2, long[][] x3) {...}
    public void M3(long[][] x3, Widget[,] x4) {...}
    public unsafe void M4(char *pc, Color **pf) {...}
    public unsafe void M5(void *pv, double *[,] pd) {...}
    public void M6(int i, params object[] args) {...}
}
}
"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"

```

● 特性和索引器。

```

namespace Acme
{
    class Widget: IProcess
    {
        public int Width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}
"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"

```

● 事件。

```

namespace Acme
{
    class Widget: IProcess
    {
        public event Del AnEvent;
    }
}
"E:Acme.Widget.AnEvent"

```

● 一元运算符。

```

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}
"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"

```

下面列出可使用的一元运算符函数名称的完整集合: op_UnaryPlus, op_UnaryNegation, op_LogicalNot, op_OnesComplement, op_Increment, op_Decrement, op_True 和 op_False。

● 二元运算符。

```

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}
"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"

```

下面列出可使用的二元运算符函数名称的完整集合：op_Addition, op_Subtraction, op_Multiply, op_Division, op_Modulus, op_BitwiseAnd, op_BitwiseOr, op_ExclusiveOr, op_LeftShift, op_RightShift, op_Equality, op_Inequality, op_LessThan, op_LessThanOrEqual, op_GreaterThan 和 op_GreaterThanOrEqual。

- 转换运算符具有一个尾随“~”，然后再跟返回类型。

```

namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}
"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"

```

A.4 示例

下面几节提供一个示例。

A.4.1 C# 源代码

下面的示例列出一个 Point 类的源代码：

```

namespace Graphics
{
    /// <remarks>Class <c>Point</c> models a point in a two-dimensional plane.
    /// </remarks>
    public class Point
    {
        /// <summary>Instance variable <c>x</c> represents the point's
        /// x-coordinate.</summary>
        private int x;
        /// <summary>Instance variable <c>y</c> represents the point's
        /// y-coordinate.</summary>
        private int y;
        /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
        public int X
        {
            get { return x; }
            set { x = value; }
        }
    }
}

```



```

    /// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
    public int Y
    {
        get { return y; }
        set { y = value; }
    }
    /// <summary>This constructor initializes the new Point to
    /// (0,0).</summary>
    public Point() : this(0,0) {}
    /// <summary>This constructor initializes the new Point to
    /// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
    /// <param><c>xor</c> is the new Point's x-coordinate.</param>
    /// <param><c>yor</c> is the new Point's y-coordinate.</param>
    public Point(int xor, int yor) {
        X = xor;
        Y = yor;
    }
    /// <summary>This method changes the point's location to
    /// the given coordinates.</summary>
    /// <param><c>xor</c> is the new x-coordinate.</param>
    /// <param><c>yor</c> is the new y-coordinate.</param>
    /// <see cref="Translate"/>
    public void Move(int xor, int yor) {
        X = xor;
        Y = yor;
    }
    /// <summary>This method changes the point's location by
    /// the given x- and y-offsets.
    /// <example>For example:
    /// <code>
    ///     Point p = new Point(3,5);
    ///     p.Translate(-1,3);
    /// </code>
    /// results in <c>p</c>'s having the value (2,8).
    /// </example>
    /// </summary>
    /// <param><c>xor</c> is the relative x-offset.</param>
    /// <param><c>yor</c> is the relative y-offset.</param>
    /// <see cref="Move"/>
    public void Translate(int xor, int yor) {
        X += xor;
        Y += yor;
    }
    /// <summary>This method determines whether two Points have the same
    /// location.</summary>
    /// <param><c>o</c> is the object to be compared to the current object.
    /// </param>
    /// <returns>True if the Points have the same location and they have
    /// the exact same type; otherwise, false.</returns>
    /// <seealso cref="operator==">
    /// <seealso cref="operator!=">
    public override bool Equals(object o) {
        if (o == null) {
            return false;
        }
        if (this == o) {
            return true;
        }
        if (GetType() == o.GetType()) {
            Point p = (Point)o;
            return (X == p.X) && (Y == p.Y);
        }
    }

```

```

    }
    return false;
}
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
/// without any leading, training, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + ", " + Y + ")";
}
/// <summary>This operator determines whether two Points have the same
/// location.</summary>
/// <param><c>p1</c> is the first Point to be compared.</param>
/// <param><c>p2</c> is the second Point to be compared.</param>
/// <returns>True if the Points have the same location and they have
/// the exact same type; otherwise, false.</returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator!=">
public static bool operator==(Point p1, Point p2) {
    if ((object)p1 == null || (object)p2 == null) {
        return false;
    }

    if (p1.GetType() == p2.GetType()) {
        return (p1.X == p2.X) && (p1.Y == p2.Y);
    }

    return false;
}
/// <summary>This operator determines whether two Points have the same
/// location.</summary>
/// <param><c>p1</c> is the first Point to be compared.</param>
/// <param><c>p2</c> is the second Point to be compared.</param>
/// <returns>True if the Points do not have the same location and the
/// exact same type; otherwise, false.</returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator==">
public static bool operator!=(Point p1, Point p2) {
    return !(p1 == p2);
}
/// <summary>This is the entry point of the Point class testing
/// program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any nontrivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // class test code goes here
}
}
}

```

A.4.2 产生的 XML

以下是文档生成器根据给定类 **Point** 的源代码（如上节所示）所产生的输出：

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Point</name>
  </assembly>

```

```

<members>
  <member name="T:Graphics.Point">
    <remarks>Class <c>Point</c> models a point in a two-dimensional
      plane.
    </remarks>
  </member>
  <member name="F:Graphics.Point.x">
    <summary>Instance variable <c>x</c> represents the point's
      x-coordinate.</summary>
  </member>
  <member name="F:Graphics.Point.y">
    <summary>Instance variable <c>y</c> represents the point's
      y-coordinate.</summary>
  </member>
  <member name="M:Graphics.Point.#ctor">
    <summary>This constructor initializes the new Point to
      (0,0).</summary>
  </member>
  <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
    <summary>This constructor initializes the new Point to
      (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
    <param><c>xor</c> is the new Point's x-coordinate.</param>
    <param><c>yor</c> is the new Point's y-coordinate.</param>
  </member>
  <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
    <summary>This method changes the point's location to
      the given coordinates.</summary>
    <param><c>xor</c> is the new x-coordinate.</param>
    <param><c>yor</c> is the new y-coordinate.</param>
    <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
  </member>
  <member
    name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
    <summary>This method changes the point's location by
      the given x- and y-offsets.
    <example>For example:
    <code>
      Point p = new Point(3,5);
      p.Translate(-1,3);
    </code>
    results in <c>p</c>'s having the value (2,8).
    </example>
    </summary>
    <param><c>xor</c> is the relative x-offset.</param>
    <param><c>yor</c> is the relative y-offset.</param>
    <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
  </member>
  <member name="M:Graphics.Point.Equals(System.Object)">
    <summary>This method determines whether two Points have the same
      location.</summary>
    <param><c>o</c> is the object to be compared to the current
      object.
    </param>
    <returns>True if the Points have the same location and they have
      the exact same type; otherwise, false.</returns>
    <seealso
      cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
    <seealso
      cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
  </member>
  <member name="M:Graphics.Point.ToString">

```

```

        <summary>Report a point's location as a string.</summary>
        <returns>A string representing a point's location, in the form
            (x,y),
            without any leading, training, or embedded whitespace.</returns>
    </member>
    <member
name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
        <summary>This operator determines whether two Points have the
            same
            location.</summary>
        <param><c>p1</c> is the first Point to be compared.</param>
        <param><c>p2</c> is the second Point to be compared.</param>
        <returns>True if the Points have the same location and they have
            the exact same type; otherwise, false.</returns>
        <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
        <seealso
cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
    </member>
    <member
name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
        <summary>This operator determines whether two Points have the
            same
            location.</summary>
        <param><c>p1</c> is the first Point to be compared.</param>
        <param><c>p2</c> is the second Point to be compared.</param>
        <returns>True if the Points do not have the same location and
            the
            exact same type; otherwise, false.</returns>
        <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
        <seealso
cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
    </member>
    <member name="M:Graphics.Point.Main">
        <summary>This is the entry point of the Point class testing
            program.
            <para>This program tests each method and operator, and
                is intended to be run after any nontrivial maintenance has
                been performed on the Point class.</para></summary>
    </member>
    <member name="P:Graphics.Point.X">
        <value>Property <c>X</c> represents the point's
            x-coordinate.</value>
    </member>
    <member name="P:Graphics.Point.Y">
        <value>Property <c>Y</c> represents the point's
            y-coordinate.</value>
    </member>
</members>
</doc>

```

附录 B 语法

此附录是主文档中描述的词法、语法，以及不安全代码的语法扩展的摘要。这里，各语法产生式是按它们在主文档中出现的顺序列出的。

B.1 词法

- input: (输入:)
 - input-section_{opt} (输入节_{可选})
- input-section: (输入节:)
 - input-section-part (输入节部分)
 - input-section input-section-part (输入节 输入节部分)
- input-section-part: (输入节部分:)
 - input-elements_{opt} new-line (输入元素_{可选} 新行)
 - pp-directive (pp 指令)
- input-elements: (输入元素:)
 - input-element (输入元素)
 - input-elements input-element (输入元素 输入元素)
- input-element: (输入元素:)
 - whitespace (空白)
 - comment (注释)
 - token (标记)

B.1.1 行结束符

- new-line: (新行:)
 - 回车符 (U+000D)
 - 换行符 (U+000A)
 - 回车符 (U+000D) 后跟换行符 (U+000A)
 - 行分隔符 (U+2028)
 - 段落分隔符 (U+2029)

B.1.2 空白

whitespace: (空白:)

任何含 Unicode 类 Zs 的字符

水平制表符 (U+0009)

垂直制表符 (U+000B)

换页符 (U+000C)

B.1.3 注释

comment: (注释:)

single-line-comment (单行注释)

delimited-comment (带分隔符的注释)

single-line-comment: (单行注释:)

// input-characters_{opt} (// 输入字符_{可选})

input-characters: (输入字符:)

input-character (输入字符)

input-characters input-character (输入字符 输入字符)

input-character: (输入字符:)

除换行符外的任何 Unicode 字符

new-line-character: (换行符:)

回车符 (U+000D)

换行符 (U+000A)

行分隔符 (U+2028)

段落分隔符 (U+2029)

delimited-comment: (带分隔符的注释:)

/* delimited-comment-characters_{opt} */ (/* 带分隔符的注释字符_{可选} */)

delimited-comment-characters: (带分隔符的注释字符:)

delimited-comment-character (带分隔符的注释字符)

delimited-comment-characters delimited-comment-character (带分隔符的注释字符 带分隔符的注释字符)

delimited-comment-character: (带分隔符的注释字符:)

not-asterisk (非星号)

* not-slash (* 非斜杠)

not-asterisk: (非星号:)

除 * 外的任何 Unicode 字符

not-slash: (非斜杠:)

除 / 外的任何 Unicode 字符

B.1.4 标记

token: (标记:)

identifier (标识符)

keyword (关键字)

integer-literal (整数)

real-literal (实数)

character-literal (字符)

string-literal (字符串)

operator-or-punctuator (运算符或标点)

B.1.5 Unicode 字符转义序列

unicode-escape-sequence: (unicode 转义序列:)

\u hex-digit hex-digit hex-digit hex-digit (\u 十六进制数字 十六进制数字 十六进制数字 十六进制数字)

\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit (\U 十六进制数字 十六进制数字 十六进制数字 十六进制数字 十六进制数字 十六进制数字 十六进制数字 十六进制数字)

B.1.6 标识符

identifier: (标识符:)

available-identifier (可用的标识符)

@ identifier-or-keyword (@ 标识符或关键字)

available-identifier: (可用的标识符:)

不是“关键字”的“标识符或关键字”

identifier-or-keyword: (标识符或关键字:)

identifier-start-character identifier-part-characters_{opt} (标识符开始字符 标识符部分字符_{可选})

identifier-start-character: (标识符开始字符:)

letter-character (字母字符)

_ (the underscore character U+005F) (下划线字符 U+005F)

identifier-part-characters: (标识符部分字符:)

identifier-part-character (标识符部分字符)

identifier-part-characters identifier-part-character (标识符部分字符 标识符部分字符)

identifier-part-character: (标识符部分字符:)

- letter-character (字母字符)
- decimal-digit-character (十进制数字字符)
- connecting-character (连接字符)
- combining-character (组合字符)
- formatting-character (格式设置字符)

- letter-character: (字母字符:)
 - 类 Lu、Ll、Lt、Lm、Lo 或 Nl 的 Unicode 字符
 - 表示类 Lu、Ll、Lt、Lm、Lo 或 Nl 的字符的 unicode 转义序列
- combining-character: (组合字符:)
 - 类 Mn 或 Mc 的 Unicode 字符
 - 表示类 Mn 或 Mc 的字符的 unicode 转义序列
- decimal-digit-character: (十进制数字字符:)
 - 类 Nd 的 Unicode 字符
 - 表示类 Nd 的字符的 unicode 转义序列
- connecting-character: (连接字符:)
 - 类 Pc 的 Unicode 字符
 - 表示类 Pc 的字符的 unicode 转义序列
- formatting-character: (格式设置字符:)
 - 类 Cf 的 Unicode 字符
 - 表示类 Cf 的字符的 unicode 转义序列

B.1.7 关键字

- keyword: one of (关键字: 下列之一)

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while

B.1.8 文本

- literal: (文本:)
 - boolean-literal (布尔值)

integer-literal (整数)
 real-literal (实数)
 character-literal (字符)
 string-literal (字符串)
 null-literal (空值)
 boolean-literal: (布尔值:)
 true
 false
 integer-literal: (整数:)
 decimal-integer-literal (十进制整数)
 hexadecimal-integer-literal (十六进制整数)
 decimal-integer-literal: (十进制整数:)
 decimal-digits integer-type-suffix_{opt} (十进制数字 整数类型后缀_{可选})
 decimal-digits: (十进制数字:)
 decimal-digit (十进制数字)
 decimal-digits decimal-digit (十进制数字 十进制数字)
 decimal-digit: one of (十进制数字: 下列之一)
 0 1 2 3 4 5 6 7 8 9
 integer-type-suffix: one of (整数类型后缀: 下列之一)
 U u L l UL Ul uL ul LU Lu lU lu
 hexadecimal-integer-literal: (十六进制整数:)
 0x hex-digits integer-type-suffix_{opt} (0x 十六进制数字 整型后缀_{可选})
 0X hex-digits integer-type-suffix_{opt} (0X 十六进制数字 整型后缀_{可选})
 hex-digits: (十六进制数字:)
 hex-digit (十六进制数字)
 hex-digits hex-digit (十六进制数字 十六进制数字)
 hex-digit: one of (十六进制数字: 下列之一)
 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
 real-literal: (实数:)
 decimal-digits . decimal-digits exponent-part_{opt} real-type-suffix_{opt} (十进制数字 . 十进制数字 指数部分_{可选} 实数类型后缀_{可选})
 . decimal-digits exponent-part_{opt} real-type-suffix_{opt} (. 十进制数字 指数部分_{可选} 实数类型后缀_{可选})
 decimal-digits exponent-part real-type-suffix_{opt} (十进制数字 指数部分 实数类型后缀_{可选})
 decimal-digits real-type-suffix (十进制数字 实数类型后缀)
 exponent-part: (指数部分:)

e sign_{opt} decimal-digits (e 符号_{可选} 十进制数字)

E sign_{opt} decimal-digits (E 符号_{可选} 十进制数字)

sign: one of (符号: 下列之一)

+

real-type-suffix: one of (实数类型后缀: 下列之一)

F f D d M m

character-literal: (字符:)

' character ' (' 字符 ')

character: (字符:)

single-character (单字符)

simple-escape-sequence (简单转义序列)

hexadecimal-escape-sequence (十六进制转义序列)

unicode-escape-sequence (unicode 转义序列)

single-character: (单字符:)

除 ' (U+0027)、\ (U+005C) 和换行符外的任何字符

simple-escape-sequence: one of (简单转义序列: 下列之一)

```
\ ' \" \\ \0 \a \b \f \n \r \t \v
```

hexadecimal-escape-sequence: (十六进制转义序列:)

`\x hex-digit hex-digitopt hex-digitopt hex-digitopt` (\x 十六进制数字 十六进制数字_{可选} 十六进制数字_{可选} 十六进制数字_{可选})

string-literal: (字符串:)

regular-string-literal (规则字符串)

verbatim-string-literal (逐字的字符串)

regular-string-literal: (规则字符串:)

" regular-string-literal-characters_{opt} " (" 规则字符串字符_{可选} ")

regular-string-literal-characters: (规则字符串字符:)

regular-string-literal-character (规则字符串字符)

regular-string-literal-characters **regular-string-literal-character** (规则字符串字符 规则字符串字符)

regular-string-literal-character: (规则字符串字符:)

single-regular-string-literal-character (单个规则字符串字符)

simple-escape-sequence (简单转义序列)

hexadecimal-escape-sequence (十六进制转义序列)

unicode-escape-sequence (unicode 转义序列)

single-regular-string-literal-character: (单个规则字符串字符:)

除 " (U+0022)、\ (U+005C) 和换行符外的任何字符

verbatim-string-literal: (逐字的字符串:)

@ "verbatim-string-literal-characters_{opt}" (@ "逐字的字符串字符_{可选}")

verbatim-string-literal-characters: (逐字的字符串字符:)
 verbatim-string-literal-character (逐字的字符串字符)
 verbatim-string-literal-characters verbatim-string-literal-character (逐字的字符串
 字符 逐字的字符串字符)
verbatim-string-literal-character: (逐字的字符串字符:)
 single-verbatim-string-literal-character (单个逐字的字符串字符)
 quote-escape-sequence (引号转义序列)
single-verbatim-string-literal-character: (单个逐字的字符串字符:)
 除 " 外的任何字符
quote-escape-sequence: (引号转义序列:)
 " "
null-literal: (空文本:)
 null

B.1.9 运算符和标点符号

operator-or-punctuator:one of (运算符或标点符号: 下列之一)
 { } [] () . , : ;
 + - * / % & | ^ ! ~
 = < > ? ++ -- && || << >>
 == != <= >= += -= *= /= %= &=
 |= ^= <<= >>= ->

B.1.10 预处理指令

pp-directive: (pp 指令:)
 pp-declaration (pp 声明)
 pp-conditional (pp 条件)
 pp-line (pp 行)
 pp-diagnostic (pp 诊断)
 pp-region (pp 区域)
pp-new-line: (pp 新行:)
 whitespace_{opt} single-line-comment_{opt} new-line (空白_{可选} 单行注释_{可选} 新行)
conditional-symbol: (条件符号:)
 除 true 和 false 外的任何标识符或关键字
pp-expression: (pp 表达式:)
 whitespace_{opt} pp-or-expression whitespace_{opt} (空白_{可选} pp 或表达式 空白_{可选})
pp-or-expression: (pp 或表达式:)
 pp-and-expression (pp 与表达式)

pp-or-expression whitespace_{opt} || whitespace_{opt} pp-and-expression (pp 或表达式 空白_{可选} || 空白_{可选} pp 与表达式)

pp-and-expression: (pp 与表达式:)

pp-equality-expression (pp 相等表达式)

pp-and-expression whitespace_{opt} && whitespace_{opt} pp-equality-expression (pp 与表达式 空白_{可选} && 空白_{可选} pp 相等表达式)

pp-equality-expression: (pp 相等表达式:)

pp-unary-expression (pp 一元表达式)

pp-equality-expression whitespace_{opt} == whitespace_{opt} pp-unary-expression (pp 相等表达式 空白_{可选} == 空白_{可选} pp 一元表达式)

pp-equality-expression whitespace_{opt} != whitespace_{opt} pp-unary-expression (pp 相等表达式 空白_{可选} != 空白_{可选} pp 一元表达式)

pp-unary-expression: (pp 一元表达式:)

pp-primary-expression (pp 基本表达式)

! whitespace_{opt} pp-unary-expression (! 空白_{可选} pp 一元表达式)

pp-primary-expression: (pp 基本表达式:)

true

false

conditional-symbol (条件符号)

(whitespace_{opt} pp-expression whitespace_{opt})((空白_{可选} pp 表达式 空白_{可选}))

pp-declaration: (pp 声明:)

whitespace_{opt} # whitespace_{opt} define whitespace conditional-symbol pp-new-line (空白_{可选} # 空白_{可选} define 空白 条件符号 pp 新行)

whitespace_{opt} # whitespace_{opt} undef whitespace conditional-symbol pp-new-line (空白_{可选} # 空白_{可选} undef 空白 条件符号 pp 新行)

pp-conditional: (pp 条件:)

pp-if-section pp-elif-sections_{opt} pp-else-section_{opt} pp-endif (pp if 节 pp elif 节_{可选} pp else 节_{可选} pp endif)

pp-if-section: (pp if 节:)

whitespace_{opt} # whitespace_{opt} if whitespace pp-expression pp-new-line conditional-section_{opt} (空白_{可选} # 空白_{可选} if 空白 pp 表达式 pp 新行 条件节_{可选})

pp-elif-sections: (pp elif 节:)

pp-elif-section (pp elif 节)

pp-elif-sections pp-elif-section (pp elif 节 pp elif 节)

pp-elif-section: (pp elif 节:)

whitespace_{opt} # whitespace_{opt} elif whitespace pp-expression pp-new-line conditional-section_{opt} (空白_{可选} # 空白_{可选} elif 空白 pp 表达式 pp 新行 条件节_{可选})

pp-else-section: (pp else 节:)

`whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt` (空白_{可选}
`# 空白可选 else pp 新行 条件节可选)`

pp-endif:

`whitespaceopt # whitespaceopt endif pp-new-line` (空白_{可选} `# 空白可选`
`endif pp 新行)`

conditional-section: (条件节:)

`input-section` (输入节)
`skipped-section` (跳过节)

skipped-section: (跳过节:)

`skipped-section-part` (跳过节部分)
`skipped-section skipped-section-part` (跳过节 跳过节部分)

skipped-section-part: (跳过节部分:)

`skipped-charactersopt new-line` (跳过字符_{可选} 新行)
`pp-directive` (pp 指令)

skipped-characters: (跳过字符:)

`whitespaceopt not-number-sign input-charactersopt` (空白_{可选} 非数字符号 输入字
`符可选)`

not-number-sign: (非数字符号:)

除 `#` 外的任何输入字符

pp-line: (pp 行:)

`whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line` (空
`白可选 # 空白可选 line 空白 行指示符 pp 新行)`

line-indicator: (行指示符:)

`decimal-digits whitespace file-name` (十进制数字 空白 文件名)
`decimal-digits` (十进制数字)
`default`
`hidden`

file-name: (文件名:)

`" file-name-characters "` (" 文件名字符 ")

file-name-characters: (文件名字符:)

`file-name-character` (文件名字符)
`file-name-characters file-name-character` (文件名字符 文件名字符)

file-name-character: (文件名字符:)

除 `"` 外的任何输入字符

pp-diagnostic: (pp 诊断:)

`whitespaceopt # whitespaceopt error pp-message` (空白_{可选} `# 空白可选`
`error pp 消息)`
`whitespaceopt # whitespaceopt warning pp-message` (空白_{可选} `# 空白可选`
`warning pp 消息)`

pp-message: (pp 消息:)
 new-line (新行)
 whitespace input-characters_{opt} new-line (空白 输入字符_{可选} 新行)
pp-region: (pp 区域:)
 pp-start-region conditional-section_{opt} pp-end-region (pp 开始区域 条件节_{可选}
 pp 结束区域)
pp-start-region: (pp 开始区域:)
 whitespace_{opt} # whitespace_{opt} region pp-message (空白_{可选} # 空白_{可选}
 region pp 消息)
pp-end-region: (pp 结束区域:)
 whitespace_{opt} # whitespace_{opt} endregion pp-message (空白_{可选} # 空白_{可选}
 endregion pp 消息)

B.2 语法

B.2.1 基本概念

namespace-name: (命名空间名称:)
 namespace-or-type-name (命名空间或类型名称)
type-name: (类型名:)
 namespace-or-type-name (命名空间或类型名称)
namespace-or-type-name: (命名空间或类型名称:)
 identifier (标识符)
 namespace-or-type-name . identifier (命名空间或类型名称 . 标识符)

B.2.2 类型

type: (类型:)
 value-type (值类型)
 reference-type (引用类型)
value-type: (值类型:)
 struct-type (结构类型)
 enum-type (枚举类型)
struct-type: (结构类型:)
 type-name (类型名称)
 simple-type (简单类型)
simple-type: (简单类型:)
 numeric-type (数值类型)

bool

numeric-type: (数值类型:)

- integral-type (整型)
- floating-point-type (浮点类型)
- decimal

integral-type: (整型:)

- sbyte
- byte
- short
- ushort
- int
- uint
- long
- ulong
- char

floating-point-type: (浮点类型:)

- float
- double

enum-type: (枚举类型:)

- type-name (类型名)

reference-type: (引用类型:)

- class-type (类类型)
- interface-type (接口类型)
- array-type (数组类型)
- delegate-type (委托类型)

class-type: (类类型:)

- type-name (类型名)
- object
- string

interface-type: (接口类型:)

- type-name (类型名)

array-type: (数组类型:)

- non-array-type rank-specifiers (非数组类型 秩说明符)

non-array-type: (非数组类型:)

- type (类型)

rank-specifiers: (秩说明符:)

- rank-specifier (秩说明符)
- rank-specifiers rank-specifier (秩说明符 秩说明符)

rank-specifier: (秩说明符)

[dim-separators_{opt}] ([维度分隔符_{可选}])
dim-separators: (维度分隔符:)
,
dim-separators , (维度分隔符 ,)
delegate-type: (委托类型:)
type-name (类型名)

B.2.3 变量

variable-reference: (变量引用:)
expression (表达式)

B.2.4 表达式

argument-list: (参数列表:)
argument (参数)
argument-list , argument (参数列表 , 参数)
argument: (参数:)
expression (表达式)
ref variable-reference (ref 变量引用)
out variable-reference (out 变量引用)
primary-expression: (基本表达式:)
primary-no-array-creation-expression (非数组创建基本表达式)
array-creation-expression (数组创建表达式)
primary-no-array-creation-expression: (非数组创建基本表达式:)
literal (文本)
simple-name (简称)
parenthesized-expression (带括号的表达式)
member-access (成员访问)
invocation-expression (调用表达式)
element-access (元素访问)
this-access (this 访问)
base-access (base 访问)
post-increment-expression (后增量表达式)
post-decrement-expression (后减量表达式)
object-creation-expression (对象创建表达式)
delegate-creation-expression (委托创建表达式)
typeof-expression (typeof 表达式)
sizeof-expression (sizeof 表达式)

checked-expression (checked 表达式)
 unchecked-expression (unchecked 表达式)
 simple-name: (简称:)
 identifier (标识符)
 parenthesized-expression: (带括号的表达式:)
 (expression) ((表达式))
 member-access: (成员访问:)
 primary-expression . identifier (基本表达式 . 标识符)
 predefined-type . identifier (预定义类型 . 标识符)
 predefined-type: one of (预定义类型: 下列之一)
 bool byte char decimal double float int long
 object sbyte short string uint ulong ushort
 invocation-expression: (调用表达式:)
 primary-expression (argument-list_{opt}) (基本表达式 (参数列表_{可选}))
 element-access: (元素访问:)
 primary-no-array-creation-expression [expression-list] (非数组创建基本表达式 [表达式列表])
 expression-list: (表达式列表:)
 expression (表达式)
 expression-list , expression (表达式列表 , 表达式)
 this-access: (this 访问:)
 this
 base-access: (base 访问:)
 base . identifier (base . 标识符)
 base [expression-list] (base [表达式列表])
 post-increment-expression: (后增量表达式:)
 primary-expression ++ (基本表达式 ++)
 post-decrement-expression: (后减量表达式:)
 primary-expression -- (基本表达式 --)
 object-creation-expression: (对象创建表达式:)
 new type (argument-list_{opt}) (new 类型 (参数列表_{可选}))
 array-creation-expression: (数组创建表达式:)
 new non-array-type [expression-list] rank-specifiers_{opt} array-initializer_{opt}
 (new 非数组类型 [表达式列表] 秩说明符_{可选} 数组初始值设定项_{可选})
 new array-type array-initializer (new 数组类型 数组初始值设定项)
 delegate-creation-expression: (委托创建表达式:)
 new delegate-type (expression) (new 委托类型 (表达式))
 typeof-expression: (typeof 表达式:)
 typeof (type) (typeof (类型))

typeof (void)

checked-expression: (checked 表达式:)

checked (expression) (checked (表达式))

unchecked-expression: (unchecked 表达式:)

unchecked (expression) (unchecked (表达式))

unary-expression: (一元表达式:)

primary-expression (基本表达式)

+ unary-expression (+ 一元表达式)

- unary-expression (- 一元表达式)

! unary-expression (! 一元表达式)

~ unary-expression (~ 一元表达式)

* unary-expression (* 一元表达式)

pre-increment-expression (前增量表达式)

pre-decrement-expression (前减量表达式)

cast-expression (强制转换表达式)

pre-increment-expression: (前增量表达式:)

++ unary-expression (++ 一元表达式)

pre-decrement-expression: (前减量表达式:)

-- unary-expression (-- 一元表达式)

cast-expression: (强制转换表达式:)

(type) unary-expression ((类型) 一元表达式)

multiplicative-expression: (乘法表达式:)

unary-expression (一元表达式)

multiplicative-expression * unary-expression (乘法表达式 * 一元表达式)

multiplicative-expression / unary-expression (乘法表达式 / 一元表达式)

multiplicative-expression % unary-expression (乘法表达式 % 一元表达式)

additive-expression: (增量表达式:)

multiplicative-expression (乘法表达式)

additive-expression + multiplicative-expression (加法表达式 + 乘法表达式)

additive-expression - multiplicative-expression (加法表达式 - 乘法表达式)

shift-expression: (移位表达式:)

additive-expression (加法表达式)

shift-expression << additive-expression (移位表达式 << 加法表达式)

shift-expression >> additive-expression (移位表达式 >> 加法表达式)

relational-expression: (关系表达式:)

shift-expression (移位表达式)

relational-expression < shift-expression (关系表达式 < 移位表达式)

relational-expression > shift-expression (关系表达式 > 移位表达式)

relational-expression <= shift-expression (关系表达式 <= 移位表达式)

relational-expression \geq shift-expression (关系表达式 \geq 移位表达式)
 relational-expression is type (关系表达式 is 类型)
 relational-expression as type (关系表达式 as 类型)
 equality-expression: (相等表达式:)
 relational-expression (关系表达式)
 equality-expression == relational-expression (相等表达式 == 关系表达式)
 equality-expression != relational-expression (相等表达式 != 关系表达式)
 and-expression: (与表达式:)
 equality-expression (相等表达式)
 and-expression & equality-expression (与表达式 & 相等表达式)
 exclusive-or-expression: (异或表达式:)
 and-expression (与表达式)
 exclusive-or-expression ^ and-expression (异或表达式 ^ 与表达式)
 inclusive-or-expression: (或表达式:)
 exclusive-or-expression (异或表达式)
 inclusive-or-expression | exclusive-or-expression (异或表达式 | 异或表达式)
 conditional-and-expression: (条件与表达式:)
 inclusive-or-expression (或表达式)
 conditional-and-expression && inclusive-or-expression (条件与表达式 && 或表达式)
 conditional-or-expression: (条件或表达式:)
 conditional-and-expression (条件与表达式)
 conditional-or-expression || conditional-and-expression (条件或表达式 || 条件与表达式)
 conditional-expression: (条件表达式:)
 conditional-or-expression (条件或表达式)
 conditional-or-expression ? expression : expression (条件或表达式 ? 表达式 : 表达式)
 assignment: (赋值:)
 unary-expression assignment-operator expression (一元表达式 赋值运算符 表达式)
 assignment-operator: one of (赋值运算符: 下列之一)
 = += -= *= /= %= &= | = ^= <<= >>=
 expression: (表达式:)
 conditional-expression (条件表达式)
 assignment (赋值)
 constant-expression: (常数表达式:)
 expression (表达式)
 boolean-expression: (布尔表达式:)

expression (表达式)

B.2.5 语句

statement: (语句:)

labeled-statement (标记语句)

declaration-statement (声明语句)

embedded-statement (嵌入语句)

embedded-statement: (嵌入语句:)

block (块)

empty-statement (空语句)

expression-statement (表达式语句)

selection-statement (选择语句)

iteration-statement (迭代语句)

jump-statement (跳转语句)

try-statement (try 语句)

checked-statement (checked 语句)

unchecked-statement (unchecked 语句)

lock-statement (lock 语句)

using-statement (using 语句)

block: (块:)

{ statement-list_{opt} } ({ 语句列表_{可选} })

statement-list: (语句列表:)

statement (语句)

statement-list statement (语句列表 语句)

empty-statement: (空语句:)

labeled-statement: (标记语句:)

identifier : statement (标识符 : 语句)

declaration-statement: (声明语句:)

local-variable-declaration ; (局部变量声明 ;)

local-constant-declaration ; (局部常数声明 ;)

local-variable-declaration: (局部变量声明:)

type local-variable-declarators (类型 局部变量声明符)

local-variable-declarators: (局部变量声明符:)

local-variable-declarator (局部变量声明符)

local-variable-declarators , local-variable-declarator (局部变量声明符 , 局部变量声明符)

local-variable-declarator: (局部变量声明符:)

identifier (标识符)
 identifier = local-variable-initializer (标识符 = 局部变量初始值设定项)
 local-variable-initializer: (局部变量初始值设定项:)
 expression (表达式)
 array-initializer (数组初始值设定项)
 local-constant-declaration: (局部常数声明:)
 const type constant-declarators (const 类型 常数声明符)
 constant-declarators: (常数声明符:)
 constant-declarators (常数声明符)
 constant-declarators , constant-declarator (常数声明符 , 常数声明符)
 constant-declarator: (常数声明符:)
 identifier = constant-expression (标识符 = 常数表达式)
 expression-statement: (表达式语句:)
 statement-expression ; (语句表达式 ;)
 statement-expression: (语句表达式:)
 invocation-expression (调用表达式)
 object-creation-expression (对象创建表达式)
 assignment (赋值)
 post-increment-expression (后递增表达式)
 post-decrement-expression (后递减表达式)
 pre-increment-expression (前递增表达式)
 pre-decrement-expression (前递减表达式)
 selection-statement: (选择语句:)
 if-statement (if 语句)
 switch-statement (switch 语句)
 if-statement: (if 语句:)
 if (boolean-expression) embedded-statement (if (布尔表达式) 嵌入语句)
 if (boolean-expression) embedded-statement else embedded-statement
 (if (布尔表达式) 嵌入语句 else 嵌入语句)
 boolean-expression: (布尔表达式:)
 expression (表达式)
 switch-statement: (switch 语句:)
 switch (expression) switch-block (switch (表达式) switch 块)
 switch-block: (switch 块:)
 { switch-sections_{opt} } ({ switch 节_{可选} })
 switch-sections: (switch 节:)
 switch-section (switch 节)
 switch-sections switch-section (switch 节 switch 节)
 switch-section: (switch 节:)

switch-labels statement-list (switch 标签 语句列表)

switch-labels: (switch 标签:)

switch-label (switch 标签)

switch-labels switch-label (switch 标签 switch 标签)

switch-label: (switch 标签:)

case constant-expression : (case 常数表达式:)

default : (default:)

iteration-statement: (迭代语句:)

while-statement (while 语句)

do-statement (do 语句)

for-statement (for 语句)

foreach-statement (foreach 语句)

while-statement: (while 语句:)

while (boolean-expression) embedded-statement (while (布尔表达式) 嵌入语句)

do-statement: (do 语句:)

do embedded-statement while (boolean-expression); (do 嵌入语句 while (布尔表达式);)

for-statement: (for 语句:)

for (for-initializer_{opt} ; for-condition_{opt} ; for-iterator_{opt}) embedded-statement
(for (for 初始值设定项_{可选} ; for 条件_{可选} ; for 迭代程序_{可选}) 嵌入语句)

for-initializer: (for 初始值设定项:)

local-variable-declaration (局部变量声明)

statement-expression-list (语句表达式列表)

for-condition: (for 条件:)

boolean-expression (布尔表达式)

for-iterator: (for 迭代程序:)

statement-expression-list (语句表达式列表)

statement-expression-list: (语句表达式列表:)

statement-expression (语句表达式)

statement-expression-list, statement-expression (语句表达式列表, 语句表达式)

foreach-statement: (foreach 语句:)

foreach (type identifier in expression) embedded-statement
(foreach (类型 标识符 in 表达式) 嵌入语句)

jump-statement: (跳转语句:)

break-statement (break 语句)

continue-statement (continue 语句)

goto-statement (goto 语句)

return-statement (return 语句)

throw-statement (throw 语句)
 break-statement: (break 语句:)
 break ;
 continue-statement: (continue 语句:)
 continue ;
 goto-statement: (goto 语句:)
 goto identifier ; (goto 标识符 ;)
 goto case constant-expression ; (goto case 常数表达式 ;)
 goto default ; (goto default ;)
 return-statement: (return 语句:)
 return expression_{opt} ; (return 表达式_{可选} ;)
 throw-statement: (throw 语句:)
 throw expression_{opt} ; (throw 表达式_{可选} ;)
 try-statement: (try 语句:)
 try block catch-clauses (try 块 catch 子句)
 try block finally-clause (try 块 finally 子句)
 try block catch-clauses finally-clause (try 块 catch 子句 finally 子句)
 catch-clauses: (catch 子句:)
 specific-catch-clauses general-catch-clause_{opt} (特定 catch 子句 常规 catch 子句_{可选})
 specific-catch-clauses_{opt} general-catch-clause (特定 catch 子句_{可选} 常规 catch 子句)
 specific-catch-clauses: (特定 catch 子句:)
 specific-catch-clause (特定 catch 子句)
 specific-catch-clauses specific-catch-clause (特定 catch 子句 特定 catch 子句)
 specific-catch-clause: (特定 catch 子句:)
 catch (class-type identifier_{opt}) block (catch (类类型 标识符_{可选}) 块)
 general-catch-clause: (常规 catch 子句:)
 catch block (catch 块)
 finally-clause: (finally 子句:)
 finally block (finally 块)
 checked-statement: (checked 语句:)
 checked block (checked 块)
 unchecked-statement: (unchecked 语句:)
 unchecked block (unchecked 块)
 lock-statement: (lock 语句:)
 lock (expression) embedded-statement (lock (表达式) 嵌入语句)
 using-statement: (using 语句:)
 using (resource-acquisition) embedded-statement (using (资源获取) 嵌入语句)

resource-acquisition: (资源获取:)

local-variable-declaration (局部变量声明)

expression (表达式)

B.2.6 命名空间

compilation-unit: (编译单元:)

using-directives_{opt} global-attributes_{opt} namespace-member-declarations_{opt} (using
指令_{可选} 全局特性_{可选} 命名空间成员声明_{可选})

namespace-declaration: (命名空间声明:)

namespace qualified-identifier namespace-body ;_{opt} (命名空间 限定标识
符 命名空间体 _{可选};))

qualified-identifier: (限定标识符:)

identifier (标识符)

qualified-identifier . identifier (限定标识符 . 标识符)

namespace-body: (命名空间体:)

{ using-directives_{opt} namespace-member-declarations_{opt} } ({ using 指令_{可选} 命
名空间成员声明_{可选} })

using-directives: (using 指令:)

using-directive (using 指令)

using-directives using-directive (using 指令 using 指令)

using-directive: (using 指令:)

using-alias-directive (using 别名指令)

using-namespace-directive (using 命名空间指令)

using-alias-directive: (using 别名指令:)

using identifier = namespace-or-type-name ;(using 标识符 = 命名空间
或类型名称 ;)

using-namespace-directive: (using 命名空间指令:)

using namespace-name ;(using 命名空间名称 ;)

namespace-member-declarations: (命名空间成员声明:)

namespace-member-declaration (命名空间成员声明)

namespace-member-declarations namespace-member-declaration (命名空间成员
声明 命名空间成员声明)

namespace-member-declaration: (命名空间成员声明:)

namespace-declaration (命名空间声明)

type-declaration (类型声明)

type-declaration: (类型声明:)

class-declaration (类声明)

struct-declaration (结构声明)

interface-declaration (接口声明)
 enum-declaration (枚举声明)
 delegate-declaration (委托声明)

B.2.7 类

class-declaration: (类声明:)
 attributes_{opt} class-modifiers_{opt} class identifier class-base_{opt} class-body ;_{opt} (特性_{可选} 类修饰符_{可选} class 标识符 类基_{可选} 类体 ;_{可选})
 class-modifiers: (类修饰符:)
 class-modifier (类修饰符)
 class-modifiers class-modifier (类修饰符 类修饰符)
 class-modifier: (类修饰符:)
 new
 public
 protected
 internal
 private
 abstract
 sealed
 class-base: (类基:)
 : class-type (: 类类型)
 : interface-type-list (: 接口类型列表)
 : class-type , interface-type-list (: 类类型 , 接口类型列表)
 interface-type-list: (接口类型列表:)
 interface-type (接口类型)
 interface-type-list , interface-type (接口类型列表 , 接口类型)
 class-body: (类体:)
 { class-member-declarations_{opt} } ({ 类成员声明_{可选} })
 class-member-declarations: (类成员声明:)
 class-member-declaration (类成员声明)
 class-member-declarations class-member-declaration (类成员声明 类成员声明)
 class-member-declaration: (类成员声明:)
 constant-declaration (常数声明)
 field-declaration (字段声明)
 method-declaration (方法声明)
 property-declaration (特性声明)
 event-declaration (事件声明)
 indexer-declaration (索引器声明)

operator-declaration (运算符声明)

constructor-declaration (构造函数声明)

destructor-declaration (析构函数声明)

static-constructor-declaration (静态构造函数声明)

type-declaration (类型声明)

constant-declaration: (常数声明:)

attributes_{opt} constant-modifiers_{opt} const type constant-declarators ; (特性_{可选} 常数修饰符_{可选} const 类型 常数声明符 ;)

constant-modifiers: (常数修饰符:)

constant-modifier (常数修饰符)

constant-modifiers constant-modifier (常数修饰符 常数修饰符)

constant-modifier: (常数修饰符:)

new

public

protected

internal

private

constant-declarators: (常数声明符:)

constant-declarator (常数声明符)

constant-declarators , constant-declarator (常数声明符 , 常数声明符)

constant-declarator: (常数声明符:)

identifier = constant-expression (标识符 = 常数表达式)

field-declaration: (字段声明:)

attributes_{opt} field-modifiers_{opt} type variable-declarators ; (特性_{可选} 字段修饰符_{可选} 类型 变量声明符;)

field-modifiers: (字段修饰符:)

field-modifier (字段修饰符)

field-modifiers field-modifier (字段修饰符 字段修饰符)

field-modifier: (字段修饰符:)

new

public

protected

internal

private

static

readonly

volatile

variable-declarators: (变量声明符:)

variable-declarator (变量声明符)

variable-declarators , variable-declarator (变量声明符 , 变量声明符)

variable-declarator: (变量声明符:)

 identifier (标识符)

 identifier = variable-initializer (标识符 = 变量初始值设定项)

variable-initializer: (变量初始值设定项:)

 expression (表达式)

 array-initializer (数组初始值设定项)

method-declaration: (方法声明:)

 method-header method-body (方法头 方法体)

method-header: (方法头:)

 attributes_{opt} method-modifiers_{opt} return-type member-name (formal-parameter
-list_{opt}) (特性_{可选} 方法修饰符_{可选} 返回类型 成员名 (形参表_{可选}))

method-modifiers: (方法修饰符:)

 method-modifier (方法修饰符)

 method-modifiers method-modifier (方法修饰符 方法修饰符)

method-modifier: (方法修饰符:)

 new

 public

 protected

 internal

 private

 static

 virtual

 sealed

 override

 abstract

 extern

return-type: (返回类型:)

 type (类型)

 void

member-name: (成员名:)

 identifier (标识符)

 interface-type . identifier (接口类型 . 标识符)

method-body: (方法体:)

 block (块)

 ;

formal-parameter-list: (形参表:)

 fixed-parameters (固定参数)

 fixed-parameters , parameter-array (固定参数 , 参数数组)

parameter-array (参数数组)

fixed-parameters: (固定参数:)

fixed-parameter (固定参数)

fixed-parameters , fixed-parameter (固定参数 , 固定参数)

fixed-parameter: (固定参数:)

attributes_{opt} parameter-modifier_{opt} type identifier (特性_{可选} 参数修饰符_{可选} 类型 标识符)

parameter-modifier: (参数修饰符:)

ref

out

parameter-array: (参数数组:)

attributes_{opt} params array-type identifier (特性_{可选} params 数组类型 标识符)

property-declaration: (特性声明:)

attributes_{opt} property-modifiers_{opt} type member-name { accessor-declarations }
(特性_{可选} 特性修饰符_{可选} 类型 成员名 { 访问器声明 })

property-modifiers: (特性修饰符:)

property-modifier (特性修饰符)

property-modifiers property-modifier (特性修饰符 特性修饰符)

property-modifier: (特性修饰符:)

new

public

protected

internal

private

static

virtual

sealed

override

abstract

extern

member-name: (成员名:)

identifier (标识符)

interface-type . identifier (接口类型 . 标识符)

accessor-declarations: (访问器声明:)

get-accessor-declaration set-accessor-declaration_{opt} (get 访问器声明 set 访问器声明_{可选})

set-accessor-declaration get-accessor-declaration_{opt} (set 访问器声明 get 访问器声明_{可选})

get-accessor-declaration: (get 访问器声明:)

`attributesopt get accessor-body` (特性_{可选} get 访问器体)
`set-accessor-declaration:` (set 访问器声明:)
`attributesopt set accessor-body` (特性_{可选} set 访问器体)
`accessor-body:` (访问器体:)
`block` (块)
`;`
`event-declaration:` (事件声明:)
`attributesopt event-modifiersopt event type variable-declarators ;` (特性_{可选} 事件
修饰符_{可选} event 类型 变量声明符 ;)
`attributesopt event-modifiersopt event type member-name { event-accessor-`
`declarations }` (特性_{可选} 事件修饰符_{可选} event 类型 成员名 { 事件访问器声
明})
`event-modifiers:` (事件修饰符:)
`event-modifier` (事件修饰符)
`event-modifiers event-modifier` (事件修饰符 事件修饰符)
`event-modifier:` (事件修饰符:)
`new`
`public`
`protected`
`internal`
`private`
`static`
`virtual`
`sealed`
`override`
`abstract`
`extern`
`event-accessor-declarations:` (事件访问器声明:)
`add-accessor-declaration remove-accessor-declaration` (添加访问器声明 移除访
问器声明)
`remove-accessor-declaration add-accessor-declaration` (移除访问器声明 添加访
问器声明)
`add-accessor-declaration:` (添加访问器声明:)
`attributesopt add block` (特性_{可选} add 块)
`remove-accessor-declaration:` (移除访问器声明:)
`attributesopt remove block` (特性_{可选} remove 块)
`indexer-declaration:` (索引器声明:)
`attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }`
(特性_{可选} 索引器修饰符_{可选} 索引器声明符 { 访问器声明 })

indexer-modifiers: (索引器修饰符:)

indexer-modifier (索引器修饰符)

indexer-modifiers **indexer-modifier** (索引器修饰符 索引器修饰符)

indexer-modifier: (索引器修饰符:)

new

public

protected

internal

private

virtual

sealed

override

abstract

extern

indexer-declarator: (索引器声明符:)

type **this** [**formal-parameter-list**] (类型 **this** [形参表])

type **interface-type** . **this** [**formal-parameter-list**] (类型 接口类型 . **this** [形参表])

operator-declaration: (运算符声明:)

attributes_{opt} **operator-modifiers** **operator-declarator** **operator-body** (特性_{可选} 运算符修饰符 运算符声明符 运算符体)

operator-modifiers: (运算符修饰符:)

operator-modifier (运算符修饰符)

operator-modifiers **operator-modifier** (运算符修饰符 运算符修饰符)

operator-modifier: (运算符修饰符:)

public

static

extern

operator-declarator: (运算符声明符:)

unary-operator-declarator (一元运算符声明符)

binary-operator-declarator (二元运算符声明符)

conversion-operator-declarator (转换运算符声明符)

unary-operator-declarator: (一元运算符声明符:)

type **operator** **overloadable-unary-operator** (**type identifier**) (类型 **operator** 可重载的一元运算符 (类型 标识符))

overloadable-unary-operator: **one of** (可重载的一元运算符: 下列之一)

+ **-** **!** **~** **++** **--** **true** **false**

binary-operator-declarator: (二元运算符声明符:)

type **operator** **overloadable-binary-operator** (**type identifier** , **type identifier**)

(类型 operator 可重载的二元运算符 (类型 标识符 , 类型 标识符))

overloadable-binary-operator: one of (可重载的二元运算符: 下列之一)

+ - * / % & | ^ << >> == != > < >= <=

conversion-operator-declarator: (转换运算符声明符:)

implicit operator type (type identifier) (implicit operator 类型 (类型 标识符))

explicit operator type (type identifier) (explicit operator 类型 (类型 标识符))

operator-body: (运算符体:)

block (块)

;

constructor-declaration: (构造函数声明:)

attributes_{opt} constructor-modifiers_{opt} constructor-declarator constructor-body (特性_{可选} 构造函数修饰符_{可选} 构造函数声明符 构造函数体)

constructor-modifiers: (构造函数修饰符:)

constructor-modifier (构造函数修饰符)

constructor-modifiers constructor-modifier (构造函数修饰符 构造函数修饰符)

constructor-modifier: (构造函数修饰符:)

public

protected

internal

private

extern

constructor-declarator: (构造函数声明符:)

identifier (formal-parameter-list_{opt}) constructor-initializer_{opt} (标识符 (形参表_{可选}) 构造函数初始值设定项_{可选})

constructor-initializer: (构造函数初始值设定项:)

: base (argument-list_{opt}) (: base (实参列表_{可选}))

: this (argument-list_{opt}) (: this (实参列表_{可选}))

constructor-body: (构造函数体:)

block (块)

;

static-constructor-declaration: (静态构造函数声明:)

attributes_{opt} static-constructor-modifiers identifier () static-constructor-body (特性_{可选} 静态构造函数修饰符 标识符 () 静态构造函数体)

static-constructor-modifiers: (静态构造函数修饰符:)

extern_{opt} static (extern_{可选} static)

static extern_{opt} (static extern_{可选})

static-constructor-body: (静态构造函数体:)

block (块)
;
destructor-declaration: (析构函数声明:)
attributes_{opt} extern_{opt} ~ identifier () destructor-body (特性_{可选} extern _{可选} ~ 标识符
() 析构函数体)
destructor-body: (析构函数体:)
block (块)
;

B.2.8 结构

struct-declaration: (结构声明:)
attributes_{opt} struct-modifiers_{opt} struct identifier struct-interfaces_{opt} struct-body ;_{opt}
(特性_{可选} 结构修饰符_{可选} struct 标识符 结构接口_{可选} 结构体 ;_{可选})
struct-modifiers: (结构修饰符:)
struct-modifier (结构修饰符)
struct-modifiers struct-modifier (结构修饰符 结构修饰符)
struct-modifier: (结构修饰符:)
new
public
protected
internal
private
struct-interfaces: (结构接口:)
: interface-type-list (: 接口类型列表)
struct-body: (结构体:)
{ struct-member-declarations_{opt} } ({ 结构成员声明_{可选} })
struct-member-declarations: (结构成员声明:)
struct-member-declaration (结构成员声明)
struct-member-declarations struct-member-declaration (结构成员声明 结构成员
声明)
struct-member-declaration: (结构成员声明:)
constant-declaration (常数声明)
field-declaration (字段声明)
method-declaration (方法声明)
property-declaration (特性声明)
event-declaration (事件声明)
indexer-declaration (索引器声明)
operator-declaration (运算符声明)

constructor-declaration (构造函数声明)
static-constructor-declaration (静态构造函数声明)
type-declaration (类型声明)

B.2.9 数组

array-type: (数组类型:)
 non-array-type rank-specifiers (非数组类型 秩说明符)
non-array-type: (非数组类型:)
 type (类型)
rank-specifiers: (秩说明符:)
 rank-specifier (秩说明符)
 rank-specifiers rank-specifier (秩说明符 秩说明符)
rank-specifier: (秩说明符:)
 [dim-separators_{opt}] ([维度分隔符_{可选}])
dim-separators: (维度分隔符:)
 ,
 dim-separators , (维度分隔符 ,)
array-initializer: (数组初始值设定项:)
 { variable-initializer-list_{opt} } ({ 变量初始值设定项列表_{可选} })
 { variable-initializer-list , } ({ 变量初始值设定项列表 , })
variable-initializer-list: (变量初始值设定项列表:)
 variable-initializer (变量初始值设定项)
 variable-initializer-list , variable-initializer (变量初始值设定项列表 , 变量初
 始值设定项)
variable-initializer: (变量初始值设定项:)
 expression (表达式)
 array-initializer (数组初始值设定项)

B.2.10 接口

interface-declaration: (接口声明:)
 attributes_{opt} interface-modifiers_{opt} interface identifier interface-base_{opt}
 interface-body ;_{opt} (特性_{可选} 接口修饰符_{可选} interface 标识符 接口基_{可选} 接口体 ;_{可选})
interface-modifiers: (接口修饰符:)
 interface-modifier (接口修饰符)
 interface-modifiers interface-modifier (接口修饰符 接口修饰符)
interface-modifier: (接口修饰符:)
 new

public
protected
internal
private

interface-base: (接口基:)
: interface-type-list (: 接口类型列表)

interface-body: (接口体:)
{ interface-member-declarations_{opt} } ({ 接口成员声明_{可选} })

interface-member-declarations: (接口成员声明:)
interface-member-declaration (接口成员声明)
interface-member-declarations interface-member-declaration (接口成员声明 接口成员声明)

interface-member-declaration: (接口成员声明:)
interface-method-declaration (接口方法声明)
interface-property-declaration (接口特性声明)
interface-event-declaration (接口事件声明)
interface-indexer-declaration (接口索引器声明)

interface-method-declaration: (接口方法声明:)
attributes_{opt} new_{opt} return-type identifier (formal-parameter-list_{opt}); (特性_{可选} new_{可选} 返回类型 标识符 (形参表_{可选});)

interface-property-declaration: (接口特性声明:)
attributes_{opt} new_{opt} type identifier { interface-accessors } (特性_{可选} new_{可选} 类型 标识符 { 接口访问器 })

interface-accessors: (接口访问器:)
attributes_{opt} get, (特性_{可选} get ;)
attributes_{opt} set, (特性_{可选} set ;)
attributes_{opt} get, attributes_{opt} set ; (特性_{可选} get ; 特性_{可选} set ;)
attributes_{opt} set, attributes_{opt} get ; (特性_{可选} set ; 特性_{可选} get ;)

interface-event-declaration: (接口事件声明:)
attributes_{opt} new_{opt} event type identifier ; (特性_{可选} new_{可选} event 类型 标识符;)

interface-indexer-declaration: (接口索引器声明:)
attributes_{opt} new_{opt} type this [formal-parameter-list] { interface-accessors }
(特性_{可选} new_{可选} 类型 this [形参表] { 接口访问器 })

B.2.11 枚举

enum-declaration: (枚举声明:)
attributes_{opt} enum-modifiers_{opt} enum identifier enum-base_{opt} enum-body ;_{opt}
(特性_{可选} 枚举修饰符_{可选} enum 标识符 枚举基_{可选} 枚举体 ;_{可选})

enum-base: (枚举基:)

: integral-type (: 整型)

enum-body: (枚举体:)

{ enum-member-declarations_{opt} } ({ 枚举成员声明_{可选} })

{ enum-member-declarations , } ({ 枚举成员声明 , })

enum-modifiers: (枚举修饰符:)

enum-modifier (枚举修饰符)

enum-modifiers enum-modifier (枚举修饰符 枚举修饰符)

enum-modifier: (枚举修饰符:)

new

public

protected

internal

private

enum-member-declarations: (枚举成员声明:)

enum-member-declaration (枚举成员声明)

enum-member-declarations , enum-member-declaration (枚举成员声明 , 枚举成员声明)

enum-member-declaration: (枚举成员声明:)

attributes_{opt} identifier (特性_{可选} 标识符)

attributes_{opt} identifier = constant-expression (特性_{可选} 标识符 = 常数表达式)

B.2.12 委托

delegate-declaration: (委托声明:)

attributes_{opt} delegate-modifiers_{opt} delegate return-type identifier (formal-parameter-list_{opt}) ; (特性_{可选} 委托修饰符_{可选} delegate 返回类型 标识符 (形参表_{可选}) ;)

delegate-modifiers: (委托修饰符:)

delegate-modifier (委托修饰符)

delegate-modifiers delegate-modifier (委托修饰符 委托修饰符)

delegate-modifier: (委托修饰符:)

new

public

protected

internal

private

B.2.13 特性

global-attributes: (全局特性:)

 global-attribute-sections (全局特性节)

global-attribute-sections: (全局特性节:)

 global-attribute-section (全局特性节)

 global-attribute-sections global-attribute-section (全局特性节全局特性节)

global-attribute-section: (全局特性节:)

 [global-attribute-target-specifier attribute-list] ([全局特性目标说明符 特性列表])

 [global-attribute-target-specifier attribute-list ,] ([全局特性目标说明符 特性列表 ,])

global-attribute-target-specifier: (全局特性目标说明符:)

 global-attribute-target : (全局特性目标 :)

global-attribute-target: (全局特性目标:)

 assembly (程序集)

 module (模块)

attributes: (特性:)

 attribute-sections (特性节)

attribute-sections: (特性节:)

 attribute-section (特性节)

 attribute-sections attribute-section (特性节 特性节)

attribute-section: (特性节:)

 [attribute-target-specifier_{opt} attribute-list] ([特性目标说明符_{可选} 特性列表])

 [attribute-target-specifier_{opt} attribute-list ,] ([特性目标说明符_{可选} 特性列表 ,])

attribute-target-specifier: (特性目标说明符:)

 attribute-target : (特性目标 :)

attribute-target: (特性目标:)

 field (字段)

 event (事件)

 method (方法)

 param (参数)

 property (特性)

 return (返回)

 type (类型)

attribute-list: (特性列表:)

 attribute (特性)

 attribute-list , attribute (特性列表 , 特性)

attribute: (特性:)
 attribute-name attribute-arguments_{opt} (特性名 特性参数_{可选})
attribute-name: (特性名:)
 type-name (类型名)
attribute-arguments: (特性参数:)
 (positional-argument-list_{opt}) ((定位参数列表_{可选}))
 (positional-argument-list , named-argument-list) ((定位参数列表 , 命名参数列表))
 (named-argument-list) ((命名参数列表))
positional-argument-list: (定位参数列表:)
 positional-argument (定位参数)
 positional-argument-list , positional-argument (定位参数列表 , 定位参数)
positional-argument: (定位参数:)
 attribute-argument-expression (特性参数表达式)
named-argument-list: (命名参数列表:)
 named-argument (命名参数)
 named-argument-list , named-argument (命名参数列表 , 命名参数)
named-argument: (命名参数:)
 identifier = attribute-argument-expression (标识符 = 特性参数表达式)
attribute-argument-expression: (特性参数表达式:)
 expression (表达式)

B.3 不安全代码的语法扩展

class-modifier: (类修饰符:)
 ...
 unsafe
struct-modifier: (结构修饰符:)
 ...
 unsafe
interface-modifier: (接口修饰符:)
 ...
 unsafe
delegate-modifier: (委托修饰符:)
 ...
 unsafe
field-modifier: (字段修饰符:)
 ...
 unsafe

method-modifier: (方法修饰符:)

...
unsafe

property-modifier: (特性修饰符:)

...
unsafe

event-modifier: (事件修饰符:)

...
unsafe

indexer-modifier: (索引器修饰符:)

...
unsafe

operator-modifier: (运算符修饰符:)

...
unsafe

constructor-modifier: (构造函数修饰符:)

...
unsafe

destructor-declaration: (析构函数声明:)

attributes_{opt} extern_{opt} unsafe_{opt} ~ identifier () destructor-body (特性_{可选}
extern _{可选} unsafe _{可选} ~ 标识符 () 析构函数体)
attributes_{opt} unsafe_{opt} extern_{opt} ~ identifier () destructor-body (特性_{可选}
unsafe _{可选} extern _{可选} ~ 标识符 () 析构函数体)

static-constructor-modifiers: (静态构造函数修饰符:)

extern_{opt} unsafe_{opt} static (extern _{可选} unsafe _{可选} 静态)
unsafe_{opt} extern_{opt} static (unsafe _{可选} extern _{可选} 静态)
extern_{opt} static unsafe_{opt} (extern _{可选} 静态 unsafe _{可选})
unsafe_{opt} static extern_{opt} (unsafe _{可选} 静态 extern _{可选})
static extern_{opt} unsafe_{opt} (静态 extern _{可选} unsafe _{可选})
static unsafe_{opt} extern_{opt} (静态 unsafe _{可选} extern _{可选})

embedded-statement: (嵌入语句:)

...
unsafe-statement (不安全语句)

unsafe-statement: (不安全语句:)

unsafe block (unsafe 块)

type: (类型:)

value-type (值类型)
reference-type (引用类型)
pointer-type (指针类型)

pointer-type: (指针类型:)
 unmanaged-type * (非托管类型 *)
 void * (**void** *)
unmanaged-type: (非托管类型:)
 type (类型)
primary-no-array-creation-expression: (非数组创建基本表达式:)
 ...
 pointer-member-access (指针成员访问)
 pointer-element-access (指针元素访问)
 sizeof-expression (**sizeof** 表达式)
unary-expression: (一元表达式:)
 ...
 pointer-indirection-expression (指针间接寻址表达式)
 addressof-expression (**addressof** 表达式)
pointer-indirection-expression: (指针间接寻址表达式:)
 * **unary-expression** (* 一元表达式)
pointer-member-access: (指针成员访问:)
 primary-expression -> **identifier** (基本表达式 -> 标识符)
pointer-element-access: (指针元素访问:)
 primary-no-array-creation-expression [**expression**] (非数组创建基本表达式 [表达式])
addressof-expression: (**addressof** 表达式:)
 & **unary-expression** (& 一元表达式)
sizeof-expression: (**sizeof** 表达式:)
 sizeof (**unmanaged-type**) (**sizeof** (非托管类型))
embedded-statement: (嵌入语句:)
 ...
 fixed-statement (固定语句)
fixed-statement: (固定语句:)
 fixed (**pointer-type** **fixed-pointer-declarators**) **embedded-statement**
 (**fixed** (指针类型 固定指针声明符) 嵌入语句)
fixed-pointer-declarators: (固定指针声明符:)
 fixed-pointer-declarator (固定指针声明符)
 fixed-pointer-declarators , **fixed-pointer-declarator** (固定指针声明符 , 固定指针声明符)
fixed-pointer-declarator: (固定指针声明符:)
 identifier = **fixed-pointer-initializer** (标识符 = 固定指针初始值设定项)
fixed-pointer-initializer: (固定指针初始值设定项:)
 & **variable-reference** (& 变量引用)

expression (表达式)

Local-variable-initializer: (变量初始值设定项:)

expression (表达式)

array-initializer (数组初始值设置项)

stackalloc-initializer (stackalloc 初始值设置项)

stackalloc-initializer: (stackalloc 初始值设定项:)

stackalloc unmanaged-type [expression] (stackalloc 非托管类型 [表达式])